BGPQ: A Heap-Based Priority Queue Design for GPUs

Yanhao Chen
yc827@cs.rutgers.edu
Rutgers, The State University of New Jersey
New Brunswick, New Jersey, USA

Fei Hua
huafei90@gmail.com
Rutgers, The State University of New Jersey
New Brunswick, New Jersey, USA

Yuwei Jin
yj243@scarletmail.rutgers.edu
Rutgers, The State University of New Jersey
New Brunswick, New Jersey, USA

Eddy Z. Zhang
eddy.zhengzhang@gmail.com
Rutgers, The State University of New Jersey
New Brunswick, New Jersey, USA

Abstract

Programming today's many-core processor is challenging. Due to the enormous amount of parallelism, synchronization is expensive. We need efficient data structures for providing automatic and scalable synchronization methods. In this paper, we focus on the priority queue data structure. We develop a heap-based priority queue implementation called BGPQ. BGPQ uses batched key nodes as the internal data representation, exploits both task parallelism and data parallelism, and is linearizable. We show that BGPQ achieves up to 88X speedup compared with four state-of-the-art CPU parallel priority queue implementations and up to 11.2X speedup over an existing GPU implementation. We also apply BGPQ to search problems, including 0-1 Knapsack and A* search. We achieve 45X-100X and 12X-46X speedup respectively over best known concurrent CPU priority queues.

CCS Concepts

• **Computing methodologies** → *Massively parallel algorithms*; **Parallel algorithms**.

Keywords

Priority Queue, GPUs, Batched Heap

ACM Reference Format:
Yanhao Chen, Fei Hua, Yuwei Jin, and Eddy Z. Zhang. 2021. BGPQ: A Heap-Based Priority Queue Design for GPUs. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472463

1 Introduction

Due to the expected tapering of transistor density, the performance improvement of modern applications must be gained from extreme parallelism in many-core processors. However, programming many-core processors requires significant time and domain expertise, including task decomposition, scheduling, and inter-thread synchronization. To enable simultaneous performance and programmability, we need efficient data structures and programming methods.

In this paper, we focus on the priority queue data structure which is a fundamental abstract data type (ADT). Priority queue plays an important role in well-known algorithm paradigms, including the Dijkstra's algorithm in graph theory, the A* search in artificial intelligence and the branch-and-bound method in combinatorics.

In order to let important applications in combinatorics, AI, and graph theory take advantage of many-core architecture, the priority queue must be implemented efficiently. There are currently very few studies on the implementations of priority queue for many-core GPU architecture. The lack of GPU priority queue studies un-matches the popularity and wide deployment of GPUs.

The heap-based priority queue implementation by He *et al.* [12] is the first generic priority queue implementation for GPUs. However, it only supports pipeline parallelism, whereas different types of parallelism can be exploited in heap-based priority queue implementations [14, 21]. John *et al.* implements the bucket heap [4] on GPUs [15]. A warp-level priority queue implementation is proposed by Crosetto [7]. Both studies focus on a specific application – the single source shortest path (SSSP) problem, instead of a generic class of applications that are built upon priority queue data structure. The GPU-friendly skip-list is implemented by Moscovici *et al.* [20] where the skip-list is partitioned into chunks and lookup is performed in parallel within each chunk. However, it does not support DeleteMin operation and is not a priority queue.

It is tempting to think that multi-core CPU priority queue implementations can be directly ported to GPUs. There are extensive studies of multi-core CPU priority queue implementations [1, 3, 6, 14, 16, 18, 21, 24, 28, 30]. But existing concurrent CPU priority queue implementations do not necessarily lend themselves to efficient GPU parallelization.

GPU architecture has two key features: (1) single instruction multiple thread (SIMT) execution model, and (2) high throughput but relatively *small* GDDR memory.

For (1), the SIMT execution model exploits both data parallelism and task parallelism. Unfortunately, most existing concurrent CPU priority queue only exploits task parallelism [1, 6, 14, 16, 18, 21, 24, 28, 30] but not data parallelism [8].

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472463

For (2), previous concurrent CPU priority queue implementations use heap [14, 21, 28], skip-list [1, 6, 16, 24], linked list [3, 30], or a combination [3, 18] thereof, as the underlying data structure. Those data structures, if directly ported to GPUs, may not fully exploit memory parallelism. Whether it is a tree-structured heap, skip-list, or linked list, memory accesses within a short time period are irregular. It cannot take advantage of high throughput memory loads/writes for coalesced accesses. Certain implementations [3, 20] trade memory space for efficiency on CPUs, which is not immediately possible for GPUs as GPU memory is scarce.

To achieve an efficient GPU priority queue implementation, one must have a deep understanding of the GPU execution and memory model, as well as a thorough understanding of many existing priority queue design choices.

In this paper, we analyze the implementation choices from the perspectives of parallelism exploitation, underlying data structures, and thread collaboration. We analyze each of these components with respect to their friendliness to the SIMT execution model and GPU memory system. By learning lessons from well established multi-core CPU priority queue implementations and taking the unique features of GPU execution into account, we propose a heap-based, lock-based, and linearizable priority queue implementation called BGPQ. The BGPQ design, even in its current simple form, can deliver significant performance gains over state-of-the-art multi-core CPU implementations and existing GPU priority queue implementation. Our main contributions are as follows:

- We present the first implementation of a *fully concurrent* **heap-based** priority queue for GPUs. It explores both data and task parallelism. The code for BGPQ is open-sourced and available on GitHub [1].
- As far as we know, BGPQ is the first **linearizable** GPU priority queue implementation and the linearizability property provides a correctness guarantee.
- We evaluate BGPQ on both synthetic data and real-world applications, including the 0-1 knapsack and the A-star search problem. We compare BGPQ with well-known CPU and GPU priority queue implementations [1, 3, 12, 16, 29], as well as Linden and Jonsson's skiplist. BGPQ can achieve 8.6X-88.3X speedup and 7.2X-11.2X speedup compared with CPU and GPU implementations, respectively.

The remainder of the paper is organized as follows. We introduce the state of the art multi-core CPU priority queue implementations, as well as their friendliness to GPU implementation in Section 2. We present our GPU priority queue implementation in Section 3 and 4. We provide a linearizability proof in Section 5. Related work is in Section 7 and evaluation results are in Section 6.

2 Anatomy of Concurrent Priority Queue Design and Optimization Choices

The priority queue ADT supports two operations: Insert, which inserts a (key, value) pair to the priority queue, and deleteMin, which returns the (key, value) pair with the smallest key from the priority queue. The concrete implementations and their associated complexity may vary depending on the internal algorithm and data representation. In the following, we discuss well-known multi-core CPU implementations and analyze their friendliness to the GPU execution model. Table 1 presents a summary of design choices for different implementations including our BGPQ implementation.

2.1 Data Structure

There are different types of underlying data structures that have been used to implement the priority queue ADT. The fundamental building blocks are heap, skip-list, and linked list. Existing concurrent priority queue implementations either use one or a combination of them. The underlying data structure stores (key, value) pairs, and the priority is associated with the key. For simplicity of discussion, we describe the operations as key insertion or deletion, whiles it refers to (key, value) pair insertion or deletion.

Heap is a tree data-structure. Using min-heap as an example, the smallest key is stored in the root node. Each node has two children nodes except the leaf nodes. A child's key is larger than or equal to its parent's key. An example is shown in Fig. 1 (a).

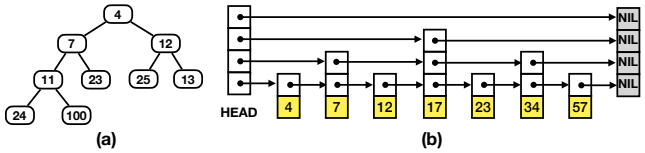


Figure 1: Heap and skip-list as underlying data structure

Heap-based priority queues are widely studied, and various versions have been proposed [9, 14, 21, 25]. It is a simple and elegant structure. It can be stored as arrays, and the index of child or parent nodes can be calculated using simple arithmetic operations. Most optimization for the concurrent heap is focused on reducing contention at the root node.

Skip-list [23] is another typical underlying data structure for priority queue. It has a probabilistically, balanced tree structure. The skip-list data structure is constructed with hierarchically ordered linked lists. The bottom layer linked list contains a sorted sequence of all keys. A key in level i also appears in level $i+1$ with probability p. p is typically set to 1/2 or 1/4. An example is shown in Fig. 1 (b).

Skip-list natively supports Insert, Search, and Delete operations. But Delete operation must know which key to delete, and it is not the same as DeleteMin operation in priority queue. The Search and Insert operations have $O(\log n)$ complexity where n is the number of keys.

Shavit and Lotan [24] implemented the first concurrent skip-list based priority queue using a two-level DeleteMin mechanism. It first identifies which key is minimal and marks this key with a *logical-delete* flag. It then physically deletes the key by searching and removing the key from corresponding location(s) at different layer(s) of the skip list. Sundell and Tsigas introduced a lock-free linearizable implementation of the skip-list [27].

Linden and Jonsson [16] further improved the skip-list implementation by batching physical deletes, which increases the number of reads but can significantly reduce the false sharing overhead caused by cache coherence protocols on CPUs. Other studies on skip-list based priority queues [6, 10] include optimizations that make the implementation non-blocking and apply flat combining as well as elimination to reduce contention.

[1] https://github.com/ruadapt/BGPQ

Table 1: A summary of priority queue implementation features, and BGPQ's comparison with existing implementations. Hunt refers to the implementation by Hunt *et al.* [14]. GFSL represents the GPU friendly skip-list implementation [20]. STSL refers to the skip-list implementation by Sundell and Tsigas [27]. LJSL refers to the implementation by Linden and Jonssan [16]. P-Sync refers the GPU implementation by He *et al.* [12]. BGPQ is our implementation.

	Hunt [14]	CBPQ [3]	STSL [27]	LJSL [16]	Spray List [1]	GFSL [20]	P-Sync [12]	BGPQ
Data Parallelism	✗	✗	✗	✗	✗	✓	✓	✓
Task Parallelism	✓	✓	✓	✓	✓	✓	✓	✓
Thread Collaboration [a]	✗	✓	✗	✗	✗	✗	✗	✓
Memory Efficient [b]	✓	✗	✗	✗	✗	✗	✓	✓
Linearizable	N/A	✓	✓	✓	N/A	N/A	N/A	✓
Data Structure	Heap	LL + SL	SL	SL	SL	SL	Heap	Heap

[a]Thread collaboration is marked if elimination or flat combining is used or thread doing delete-min steal keys from thread doing insertion in progress
[b]An implementation is efficient if and only if it uses k+$O(1)$ memory, where k is the number of keys.

Alistarh *et al.* [1] invented a relaxed priority queue implementation based on skip-list, called SprayList. The DELETE in SprayList operation returns a key among the top $O(p * log^3 p)$ keys while p is the number of threads in the system. SprayList reduces the contention for deleting the smallest key.

There are also other alternative data structures. Zhang *et al.* [30] proposed a multi-dimensional linked list. Braginsky *et al.* proposed CBPQ [3], a linked chunk-list based priority queue, which performs well under high contention.

GPU Friendliness: Both heap and skip-list have $O(\log n)$ computation complexity, but considering memory management, we believe heap is more favorable than skip list for two reasons. First, the skip-list needs more memory space than the heap. Skip-list needs to store keys (or pointers to them) that appear at different layers of skip-list. With $p = 50\%$, skip-list may use as much as twice memory as a heap. GPU memory has more than 10X throughput than CPU memory, but GPU memory resource is scarce. If running out of memory on GPU, it needs to transfer data back and forth between CPU and GPU, causing a memory bottleneck. It then defeats the purpose of application performance accelerating on GPUs.

Second, skip-list uses a linked list, which requires dynamically allocation and reclamation of memory. Current GPUs do not have as good dynamic memory management as that on CPUs. Moscovici *et al.* [20] uses a simplified array-based memory reclamation for chunked skip-list, but dynamic memory management itself is non-trivial to be parallelized on GPUs for general cases.

2.2 Parallelism Exploitation

Both heap and skip-list based priority queue implementations use fine-grained locks. Fine-grained lock gives rise to fine-grained parallelism. Each key is protected by one lock. When a key is locked by one operation, it cannot be used by another operation. Operations on different keys with no mutual exclusion can happen in parallel. In Table 1, we list the parallelism types for existing well-known priority queue implementations.

GPU Friendliness: However, most existing approaches exploit only task parallelism. To maximize performance gains on GPU, one must exploit both data and task parallelism.

GPU, as well as other computational accelerators, are equipped with hierarchical parallelism. At the coarse-grained level, it is task parallelism. At the fine-grained level, it is data parallelism. GPU hardware is made of multiple streaming multi-processors. Each multi-processor only has data parallelism. Cores within a data-parallel multi-processor must execute in lock steps. Executing different control branches will cause serialization within one streaming multi-processor, called thread divergence [31]. Each data-parallel multi-core processor has at least 32 cores. Hence the slowdown could be up to 32X if the data parallelism is not well utilized.

In existing concurrent priority queue implementations, an INSERT, DELETEMIN, or LOOKUP, has to take a tree traversal path in the heap or skip-list. Different operations typically take different traversal paths, causing control flows and thread divergence.

Very few studies for concurrent CPU implementations have exploited data parallelism. Deo and Prasad [8] proposed the only concurrent CPU heap implementation that we know explicitly exploits data parallelism. It expands a node to store multiple keys. Thus when moving a node up and down the heap, it moves multiple keys at once. He *et al.* [12] developed a natural extension of Deo and Prasad's work [8] to GPUs. However, their method requires to insert or delete a fixed number of keys at once. Moreover, it only exploits pipeline parallelism and requires a barrier between every two pipeline stages. Our BGPQ performance is much better than this implementation, as shown in Section 6.

CBPQ uses a chunked linked list. There is data parallelism for lookups within a chunk, but the CPU implementation does not exploit it. The GPU skip-list by Moscovici *et al.* [20] is an extension to the chunked link list idea. It exploits data parallelism in the SEARCH operation. It partitions each linked list into chunks that consist of multiple keys. When looking up a key, it lets threads in a warp check different elements.

2.3 Thread Collaboration

Existing studies have proposed different ways to let threads collaborate rather than contend for shared resources. Nageshwara and Prasad [21] let a DELETEMIN thread steal a key an active INSERT thread is holding to refill the root node.

Flat combining is a technique that combines multiple requests and let them be handled by one thread. Flat combining is used by Calciu *et al.* [6] for batching multiple DELETEMIN requests in the concurrent priority queue. It is also used by Braginsky *et al.* [3] for collaborating on rebuilding the first chunk in the chunked linked list of CBPQ.

Elimination is a technique that matches inverse requests in a short time. It was originally introduced to optimize concurrent stacks. It was used for the first time for concurrent priority by Calciu [6], where an elimination layer is used in combination with skip-list. It is also used in CBPQ by Braginsky *et al.* [3], where keys are stored in chunks, including the buffer for insert-small operations, making the elimination technique naturally fit into the underlying data structure.

GPU Friendliness: Thread collaboration has been introduced to help scale up priority queue performance when the contention is high. For GPUs, the number of threads running concurrently is high. It is important to have thread collaboration features while in the meantime avoiding problems such as thread divergence, memory un-coalescing, and memory resource contention.

3 BGPQ Algorithm Overview

We give an overview of the BGPQ algorithm from the perspective of data structure, parallelism, and thread collaboration. The detailed algorithm is introduced in Sec. 4.

3.1 Data Structure

We use heap as the underlying data structure, but we extend it to enable efficient parallelization and thread collaboration. Each node in the extended heap contains k keys ($k > 1$) except the root node and the buffer node. The root node contains $\leq k$ keys. The buffer node contains $\leq k-1$ keys. For each node that is not the root node or the buffer node, the node's smallest key is larger than or equal to the largest key of its parent node. The smallest key of the buffer node is larger than or equal to the largest of the root node. An example of BGPQ with $k = 4$, together with its basic operations are shown in Fig. 2.

3.2 Basic Operations

Our INSERT API supports the insertion of 1 to k keys to the heap. Our DELETEMIN API supports the deletion of 1 to k smallest keys from the heap. If there are not enough keys as requested, then all keys will be deleted from the heap.

We use the buffer node to keep track of the keys that need to be inserted into the heap but have not accumulated to k keys to form a batch. We denote this buffer node as *pBuffer*.

We show the workflow of insert operations in Fig. 3. Before the insertion-keys are placed into the *pBuffer* node, they are merged with the root node, such that the root node attains the smallest $|root|$ keys of the merged keys, where $|root|$ is the number of keys originally in the root node. The remaining keys are placed in *pBuffer*.

When the buffer node *pBuffer* overflows, it triggers a full insert-heapify process in a top-down manner. When *pBuffer* does not overflow, the nodes that are not root or pBuffer will not be modified. With the *pBuffer*, it is as if batching multiple insertion requests while still ensuring the root node contains the smallest keys.

We show the idea of deleteMin operations in Fig. 4. A DeleteMin operation extracts up to k keys at once. Assuming it wants to extract $m \leq k$ keys. If the root node has more than m keys, it does not trigger a full delete-heapify process. If the root node has less than m keys, it starts the heapify process. It first extracts the keys from the last node of the heap, merges them with those in the *pBuffer* node, obtains the k smallest ones to place them into the root node, then starts a top-down heapify process. Before the root node is unlocked, the smallest k keys of the root node and root node's child nodes are placed in the root node, part of which will be added to the retrieved-key set if the original root node has $< m$ keys.

Our implementation encourages thread collaboration in a similar way as *elimination* [6]. It is possible that within a short time, the minimal key(s) inserted by one operation can be fetched by a later delete-min request. Therefore, it does not have to access the large heap body below the root node.

We also implement another thread collaboration optimization [21] where a delete-min thread steals the insertion-keys from another thread that is performing insertion actively and use them to refill the root node.

3.3 Parallelism

Our design exploits both data and task parallelism. Node merging, sorting, and swapping can take advantage of data parallelism. In the classical heap, the basic operation between two nodes is the comparison and swap of two keys. Here, it becomes the comparison, merging, sorting, and swapping of two sets of keys, the techniques of which have already been well optimized for GPUs.

Our design also exploits memory parallelism. We store the heap as an array. Each batch node is stored in aligned consecutive memory blocks. When loading a batch node, consecutive memory blocks are loaded, and thus the memory throughput is maximized.

Task parallelism is exploited for operations on different nodes. Each node is protected by one unique lock except the root and pBuffer node. The root node and pBuffer node share one lock. Any heapify algorithm that exploits task parallelism for single-key node heap can be applied to our extended heap. We let both insertions and deletions be top-down during their tree traversal process.

We also implemented an existing approach to reduce root node contention for task parallelism similar to that for a single-key node by Hunt *et al.* [14]. The performance is similar to that of the simple top-down approach (Sec. 6).

4 Implementation

Our implementation adopts the top-down tree traversal mechanism for INSERT. For DELETEMIN operation, after the keys from the root are extracted, a top-down re-heapify process is triggered to make sure the keys in the heap still satisfy the heap property. This is a strikingly simple implementation, however, the performance gains are significant as shown in our experiments.

Each node is protected by one unique lock except the root node and the buffer node which share a lock. we let every heap node be associated with a state. The state of a node can be one of the four: AVAIL, EMPTY, TARGET, and MARKED. AVAIL and EMPTY are used to represent whether a node contains keys or not. TARGET and MARKED are used for the thread collaboration and we will discuss this later. The state of a node is protected by the corresponding lock and can only be changed when the node is locked.

We use the **SORT_SPLIT** operation on heap nodes in our implementation. Here we formally define the **SORT_SPLIT** operation between two sorted nodes:

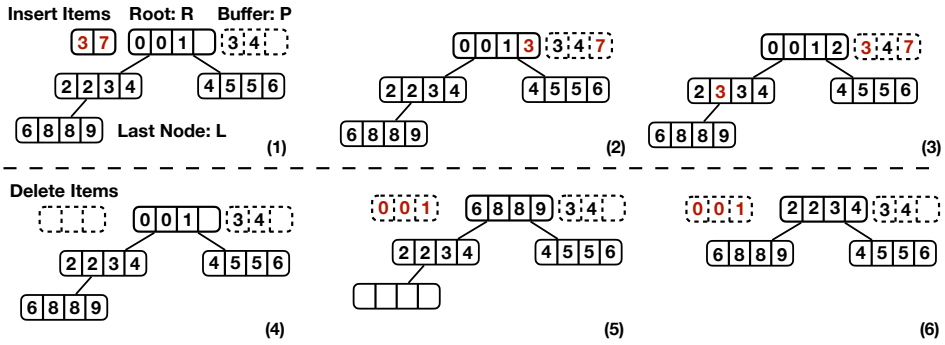


Figure 2: Underlying Data Structure for BGPQ

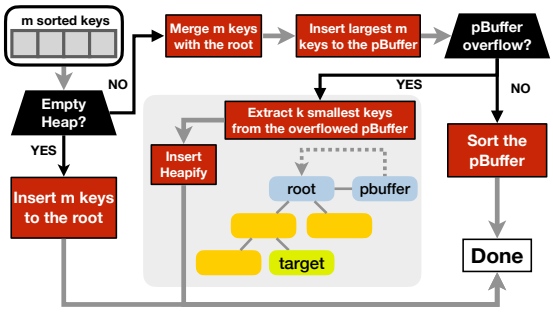


Figure 3: BGPQ Insert Operations

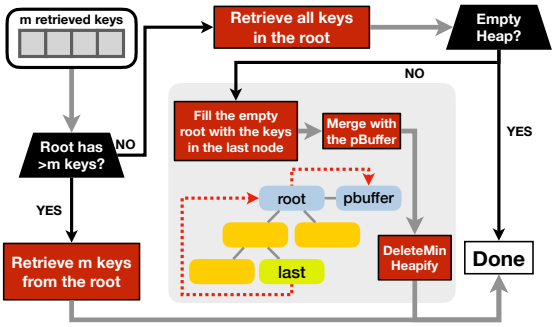


Figure 4: BGPQ DeleteMin Operations

$$(\mathbf{X}[1:M_a], \mathbf{Y}[1:M_b]) \leftarrow \textbf{Sort\_Split}(\mathbf{Z}, N_a, \mathbf{W}, N_b, M_a)$$

$$\textbf{s.t. } (\mathbf{X}, \mathbf{Y}) = sorted(\mathbf{Z}, \mathbf{W})$$

$$M_a + M_b = N_a + N_b, \max_{i=1..M_a} \mathbf{X}[i] \leq \min_{j=1..M_b} \mathbf{Y}[j]$$

$$\forall i \in [1, M_a) : \mathbf{X}[i] \leq \mathbf{X}[i+1], \forall i \in [1, M_b) : \mathbf{Y}[i] \leq \mathbf{Y}[i+1]$$

This **Sort_Split** operation returns two nodes X and Y with size M_a and M_b while $M_a + M_b$ is equal to $N_a + N_b$ while N_a and N_b are the numbers of keys in $\mathbf{Z}$ and $\mathbf{W}$ respectively. X stores the sorted M_a smallest keys and Y stores the sorted M_b largest keys from $\mathbf{Z}$ and $\mathbf{W}$. The most common value of N_a, N_b, M_a and M_b is the node capacity K which represent a **Sort_Split** operation between two full nodes. In our pseudo code if the range is not specified, it means the **Sort_Split** is performed on two full nodes.

Algorithm 1 BGPQ Insertion

```
1:  procedure BGPQ_Insert(items[], size)
2:    items[1:size] ← Sort(items[1:size]);
3:    LOCK(root);
4:    if Partial_Insert(&items, size) then return ;
5:    heapSize ← heapSize + 1; tar ← heapSize;
6:    LOCK(tar); Heap[tar].state ← TARGET; UNLOCK(tar);
7:    cur ← Insert_Heapify(NEXT(root, tar), tar, &items);
8:    LOCK(tar); UNLOCK(PARENT(cur));
9:    if Heap[tar].state = TARGET then
10:     Heap[tar] ← items; Heap[tar].state ← AVAIL;
11:   else                          ▷ A DeleteMin operation is waiting at the root.
12:     Heap[root] ← items; |root| ← K;
13:     Heap[root].state ← AVAIL; Heap[tar].state ← EMPTY;
14:   UNLOCK(tar);
15: procedure Partial_Insert(items[], size)
16:   if heapSize = 0 then                                  ▷ Heap is empty.
17:     Heap[root][1:size] ← items[1:size]; |root| ← size;
18:     Heap[root].state ← AVAIL; heapSize ← heapSize + 1;
19:     UNLOCK(root); return True;
20:   (Heap[root][1:|root|], items[1:size]) ←
          Sort_Split(Heap[root], |root|, items, size, |root|);
21:   if |pBuffer| + size < K then                          ▷ Buffer does not overflow.
22:     pBuffer[|pBuffer|+1:|pBuffer|+size] ← items[1:size];
23:     |pBuffer| ← |pBuffer| + size;
24:     UNLOCK(root); return True;
25:   else                                                  ▷ Buffer overflows.
26:     Sort(pBuffer[1:size]);
27:     (items[1:K], pBuffer[1:|pBuffer|+size-K]) ←
            Sort_Split(items, size, pBuffer, |pBuffer|, K);
28:     |pBuffer| ← |pBuffer|+size-K;
29:     return False;
30: procedure Insert_Heapify(cur, tar, items[])
31:   if cur = tar or Heap[tar].state = MARKED then return cur;
32:   LOCK(cur); UNLOCK(PARENT(cur));
33:   (Heap[cur], items) ← Sort_Split(Heap[cur], items);
34:   return INSERT_HEAPIFY(NEXT(cur, tar), tar, &items);
```

Splitting a short sequence or merging two sorted sequences can be efficiently implemented on GPUs. It can take advantage of shared memory due to the small size of the sequence. There are different existing parallel sorting primitives for GPUs including *bitonic sort*, *merge sort*, and *radix sort*. We used the GPU merge path (merging) algorithm [11] and bitonic sorting [22] in our implementation.

4.1 Insert Operation

The Insert pseudo-code is presented Alg. 1. The inserted keys are sorted first and then the root is locked (line 2-3). **Partial_Insert**() is called to insert keys, either into the buffer or as a new heap node. If the heap is empty, inserted keys *"items"* are directly placed in the

root (line 16-19), otherwise, a Sort_Split operation is performed between the root node and *"items"* to place the smaller keys in the root node (line 20). We then check if placing all inserted keys will overflow the buffer, if not, updated *"items"* are inserted into the buffer (line 21-24). Otherwise, a full heap node with k smallest keys is extracted and it invokes an insert heapify process (line 26-29).

With the help of the partial buffer, the number of insert-heapify processes is reduced. We can wait for more Insert operations until the buffer overflows and then invoke an insert-heapify process. The insert-heapify process needs to propagate k keys down to the target node. The target's state will be changed to TARGET for the thread collaboration (line 6). **Insert_Heapify**() is called at line 7 for the insert-heapify process.

The heapify process traverses the path from the root node to the target node in the tree. Each node on the path is locked before its parent node's lock is released (line 32). A Sort_Split operation is performed between the insert-keys and the current node (line 33). During the insert-heapify process, the target node's state is checked repeated. If the state has been changed to MARKED, it will break out the heapify process immediately and collaborate with the Delete-Min operation (line 11-13, 31).

When the heapify operation encounters the target node, it locks the target node and checks the state of the target node. When the target node is locked, no other operation can change the state. If the target node's state is still TARGET, it will place the inserted keys into the target node, change its state to AVAIL and release the lock (line 9-10, 14).

4.2 DeleteMin Operation

The pseudo code for DeleteMin operation of BGPQ is shown in Alg. 2 and 3. A single DeleteMin operation can retrieve up to k keys from the heap. **Partial_DeleteMin**() is called to delete keys from the root node (line 3). If the heap is already empty, we just release the root (line 16-17).

If there are enough keys in the root, the keys will be extracted (line 18-20) using **Extract_Root**() function. If the root does not contain enough keys and there are no more full nodes in the heap (line 23), we use the keys in the buffer to fill the DeleteMin operation. It is also possible that the number of keys requested cannot be satisfied. In this case, we delete all the keys in the heap and update the delete size (line 24-29). When there are still full heap nodes, we change the root node's state to EMPTY and invoke a DeleteMin-heapify process (line 31).

Since the root node is empty, Delete-Min operation checks the target node's state to see whether there is an opportunity to collaborate with the Insert operation (line 7-9). After the root node is refilled , it merges with the buffer to make sure all keys in the root node are smaller than those in the buffer (line 13). Then the heapify process **DeleteMin_Heapify** is called. The *items* and *remainedSize* are used to fulfill the remaining delete keys when the root node is updated during the heapify process (line 4, 14).

In Alg. 3, during the heapify process, both children of the current node are locked and a Sort_Split operation will be performed on the two child nodes first (line 10). Depending on which of the left and the right child originally has a larger largest key, we say node x, the largest k keys of the Sort_Splitresult will be placed in node x. Then another Sort_Split operation will be performed on the current node and the other child node y such that the current node carries the smallest k keys and the child node y carries the largest k keys. The current node is released after that. These steps repeat (line 15) until the heap property is satisfied (line 4). It is possible that the left child and the right child's states are TARGET, but as nodes with TARGET status do not carry any keys, thus the heap properties are automatically satisfied.

Algorithm 2 BGPQ DeleteMin

```
1:  procedure BGPQ_DeleteMin(items[], size)
2:    LOCK(root);
3:    if Partial_DeleteMin(&items, &size) then return ;
4:    remainedSize ← size − |items|;
5:    tar ← heapSize; heapSize ← heapSize − 1;
6:    LOCK(tar);
7:    if Heap[tar].state = TARGET then
8:        Heap[tar].state ← MARKED; UNLOCK(tar);
9:        while Heap[root].state ≠ AVAIL;
10:   else
11:       Heap[root] ← Heap[tar]; Heap[tar].state ← EMPTY;
12:       UNLOCK(tar); Heap[root].state ← AVAIL; |root| ← K
13:   (Heap[root][1:K], pBuffer[1:|pBuffer|]) ←
              Sort_Split(Heap[root], K, pBuffer, |pBuffer|, K);
14:   DeleteMin_Heapify(root, &items, remainedSize);
15: procedure Partial_DeleteMin(items[], size)
16:   if heapSize = 0 then                          ▷ Nothing to delete.
17:       size ← 0; UNLOCK(root); returnTrue;
18:   else if size < |root| then                    ▷ Root has enough keys to delete.
19:       Extract_Root(&items, size);
20:       UNLOCK(root); returnTrue;
21:   else                                          ▷ Root needs to be refilled.
22:       items[1:|root|] ← Heap[root][1:|root|]; |root| ← 0
23:       if heapSize = 1 then                      ▷ Fill the root with the buffer.
24:           |root| ← |pBuffer|; |pBuffer| ← 0;
25:           Sort(pBuffer[1:|root|]);
26:           Heap[root][1:|root|] ← pBuffer[1:|root|]
27:           size ← min(|items| + |root|, size);
28:           Extract_Root(&items, size - |items|);
29:           UNLOCK(root); returnTrue;
30:       else                                      ▷ Fill the root with a heap node.
31:           Heap[root].state ← EMPTY; returnFalse;
32: procedure Extract_Root(items[], s)
33:   items[|items|+1:|items|+s] ← Heap[root][1:s];
34:   Heap[root][1:|root|-s] ← Heap[root][s+1:|root|];
35:   |root| ← |root|-s;
```

Algorithm 3 BGPQ DeleteMin Heapify Process

```
1:  procedure DeleteMin_Heapify(cur, items[], remainedSize)
2:    (l, r) ← (LEFT(cur), RIGHT(cur));
3:    LOCK(l); LOCK(r); maxcur ← max(Heap[cur]);
4:    if maxcur ≤ min(min(Heap[l]), min(Heap[r])) then
5:        if cur = root then
6:            Extract_Root(&items, remainedSize);
7:        UNLOCK(cur); UNLOCK(l); UNLOCK(r);
8:        return ;
9:    (x, y) ← max(Heap[l]) > max(Heap[r]) ? (l, r) : (r, l);
10:   (Heap[y], Heap[x]) ← Sort_Split(Heap[l], Heap[r]);
11:   UNLOCK(x);
12:   (Heap[cur], Heap[y]) ← Sort_Split(Heap[cur], Heap[y]);
13:   if cur = root then Extract_Root(&items, remainedSize);
14:   UNLOCK(cur);
15:   DeleteMin_Heapify(y, &items, remainedSize);
```

4.3 Thread Collaboration

There are different types of thread collaborations being applied to BGPQ. As we mentioned before, an insertion thread can collaborate

with a concurrent DeleteMin thread in BGPQ. We support this collaboration using the two special states TARGET and MARKED.

When the root node needs to be refilled, the DeleteMin thread checks whether the state of the target node is TARGET or AVAIL. If it is TARGET, which means there is a chance for thread collaboration, the Delete-Min operation will change the target state to MARKED to notify the insert operation. The insert operation will be responsible for moving its insert keys to fill the root node and change the root's state to AVAIL. This thread collaboration was introduced in [21] and fits well with BGPQ.

There are also collaborations between multiple insertion threads in BGPQ. An insertion will first try to insert keys into the partial buffer. Multiple insertion operations may trigger only one insert-heapify process, that is, when the buffer overflows. Since the insert-heapify process is more expensive than only updating the root/buffer node, such collaboration can reduce the heap overhead.

Similar collaboration idea is also applied to DeleteMin threads. In BGPQ, deleting a small number of keys from the heap may not trigger an expensive DeleteMin heapify process. It waits until the DeleteMin thread requires more keys than the root contains. Also, a DeleteMin thread with the heapify process being invoked will refill the root node, so upcoming DeleteMin threads could potential take the benefit and extract these previously fetched keys.

5 Linearizability

In our implementation, an operation, whether insertion or deleteMin, has to lock the root first. We let a linearization point be placed in between the time the root is locked and unlocked by an operation.

We use the following notations. An insertion (ins) or deleteMin (del) operation takes a certain amount of time to complete. We denote the starting time as the *invocation* time, the completion as the *response* time.

We denote an operation with a 4-tuple followed by two parameters $op[s, acR, reR, t]\ (x)\ T$. The symbol op is the type of the operation: *ins* or *del*. s is the invocation time, t is the response time. acR refers to the time the root is locked; reR refers to the time the root is unlocked. x refers to the set of keys to be inserted or to be extracted by the operation op.

A concurrent history H with n operations is denoted as:

$$H = \{op_i\ [s_i, acR_i, reR_i, t_i]\ (x_i)\ T_i |\ 1 \le i \le n\ \} \tag{1}$$

We impose a total ordering of these events with respect to the lock/unlock of the root. For the u-th and v-th operations, $op_u\ [s_u, acR_u, reR_u, t_u]\ (x_u)\ T_u$, and $op_v\ [s_v, acR_v, reR_v, t_v]\ (x_v)\ T_v$, we have $u < v$ if and only if $reR_u < acR_v$.

We let I_i and D_i respectively denote the set of keys inserted and deleted preceding and including the i-th operation in H.

$$I_i = \bigcup_{op_m = ins} x_m,\ and\ D_i = \bigcup_{op_n = del} x_n$$

Linearizability We construct a sequential history with respect to the concurrent history and show that the sequential history is valid. An important variant of our algorithm is that (1) if a node (not including the buffer node) is not locked, its key values are equal to or less than the keys in all its descendent in the heap, and (2) the root's keys are always smaller than or equal to that in the buffer node. It is because we use fined-grained locks. Each insertion operation keeps a node locked until it has made sure the node have smaller keys or equal keys than the keys it originally contains. Each *del* operation will not unlock a node or its child node(s) until the node has been made smaller or equal to its child node(s).

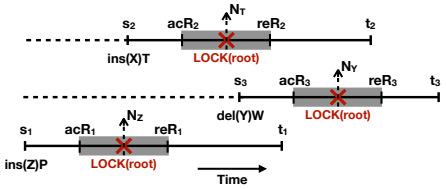


Figure 5: BGPQ linearization points

Proof. The sequential history is constructed as follows. We construct n operations, and each one of them takes effect instantaneously. We let N_i denote the i-th linearization point, an arbitrary point between the root locking time acR_i and unlocking time reR_i of an operation in H as shown in Fig. 5.

$$S = \{op_i^S\ [N_i]\ (x_i^S)\ |\ 1 \le i \le n,\ acR_i < N_i < reR_i\} \tag{2}$$

We let $op_i^S = op_i$. We let $x_i^S = x_i$ if $op_i = ins$. For each *del* operation, it extracts up to the same number of keys as in the concurrent history, $|x_i^S| = |x_i|$, if there are at least $|x_i|$ keys in the heap with respect to the execution induced by the sequential history. Later, we show $|x_i^S| = |x_i|$ always holds.

I_i^S and D_i^S are defined in a similar way as I_i and D_i for S in Equation (2). We let $I_0 = I_0^S = \emptyset$ and $D_0 = D_0^S = \emptyset$. We prove that $D_i = D_i^S$, $I_i = I_i^S$, and $x_i = x_i^S$ by induction. Assuming that up to the k, $D_k = D_k^S$, $I_k = I_k^S$, and $x_i = x_i^S$. We now prove that $D_{k+1} = D_{k+1}^S$, $I_{k+1} = I_{k+1}^S$. There are two cases.

Case I – the (k+1)-th operation is an insertion:
$ins[s_{k+1}, acR_{k+1}, reR_{k+1}, t_{k+1}]\ (x_{k+1})\ T_{k+1}$. Right after thread T_{k+1} unlocks the root in our implementation, it must have already merged the keys with the root and the buffer node. See the code line 21 and 29 in Algorithm 1. An insertion of k keys will propagate to the corresponding target location if necessary, after the root is unlocked by thread T_{k+1}.

Therefore, the heap contains the keys of $I_k - D_k + x_{k+1}$. We have $I_{k+1} = I_k \cup x_{k+1}$. It holds for both the sequential execution case too such that $I_{k+1}^S = I_{k+1}^S$. The D sets remain unchanged, $D_{k+1} = D_k$ and hence $D_{k+1}^S = D_k^S$.

Since originally the root's keys are smaller than or equal to all other keys in the heap, the new $|root|$ keys are smaller than the original root and the newly inserted keys, and hence smaller than or equal to all keys in the heap.

Case II – If the (k+1)-th operation is a *del* operation, it aims to retrieve $r \ge$ keys, to be stored in the return ndoe x_{k+1} (if the heap contains r keys, it will be $|x_{k+1}|$ keys).

In our implementation, right after thread T_{k+1} has locked the root, if the root has more than r keys, then T_{k+1} will just retrieve k smallest keys from the root, and unlock the root (line 21 in Algorithm 2). Since the root contains the smallest $|root|$ keys of all keys in the heap, the r smallest keys in the root will be the r smallest keys of the entire heap.

If the root has less than r keys, our implementation will lock the left and right child (if any) of the root, extract keys from the last

node in the heap (if any), fill them in the root. Now before the root is unlocked, it finds $r - |root|$ smallest keys among the following nodes: the root, the child node(s) of the root, and the buffer node, if they have at least $r - |root|$ keys. If they do not contain $r - |root|$ nodes, then retrieve all the keys. Since root's child node(s), if not locked, contains keys that are smaller than the rest of the heap, then $r - |root|$ smallest keys of these nodes are the $r - |root|$ smallest of all. Therefore, $x_{k+1} = min_r(I_k - D_k)$, or $x_{k+1} = I_k - D_k$ if the heap has less than r keys.

It is trivial to show that in the sequential history S, $x^S_{k+1} = min_r(I^S_k - D^S_k)$ or $x^S_{k+1} = I^S_k - D^S_k$ if the heap has less than r keys. Hence, we have proved $x^S_{k+1} = x_{k+1}$, and $D^S_{k+1} = D_{k+1}$. The set of keys in I remain unchanged, so $I^S_{k+1} = I_{k+1}$.

By induction, $I_i = D_i$, and $x^S_{i+1} = x_{i+1}$, for $1 \le i \le n$. It is proved that BGPQ is linearizable. [2] □

6 Evaluation

6.1 Experiment Setup

We compare BGPQ with four CPU baselines: TBB [29], the priority queue in Intel Threading Building Blocks, which is a widely used C++ library for shared memory parallel programming; CBPQ [3] , a chunk based lock-free parallel priority queue implementation; LJSL [16], a skip-list based concurrent priority queue and SprayList [1], a relaxed skip-list based priority queue implementation. We also compare one GPU baseline [12] which we refer to as P-Sync.

We use an NVIDIA TITAN X GPU with 28 streaming multi-processors (SMPs), each with 128 cores. The maximum number of active threads per SMP is 2048. We use CUDA 10.0. In the following experiments, we use the configuration of 128 thread blocks per kernel, 512 threads per block, and 1024 keys per batch, for both BGPQ and P-Sync.

CPU baselines are evaluated on a server of four Intel Xeon E7-4870 processors with 1TB memory. Each processor has 10 cores with a 2.40 GHz working frequency and can multiplex two hardware threads. We used 80 threads in our experiments. All these four priority queues were implemented in C++ and compiled with –O3 optimization level.

6.2 BGPQ Design Choice

We first discuss the design choice of BGPQ. We vary the thread block size, node capacity, and the number of thread blocks for different experiments. We show the performance of inserting 64M random 32-bit keys into an empty heap and then deleting all keys in Fig. 6.

BGPQ node capacity: Fig. 6a and 6b shows the performance with varying BGPQ node capacity and thread block size. Due to the limits in shared memory size per thread block, the maximum batch size we used is 1K. When thread block size is the same, for both Insert and DeleteMin operations, we can observe that the performance becomes better when the node capacity is larger as it provides more intra-node parallelism. The results also show that it is not always good to increase the thread block size. A large thread block size can increase the overhead of synchronization within a thread block. Among all these configurations, we choose one with thread block size = 512 and node capacity = 1k for later experiments as it has the best performance for both Insert and DeleteMin operations.

Thread block number: The more thread blocks, the more concurrency we can gain, and also, the more contention on the heap. The experiment used 512 threads in a thread block with a 1K node capacity. The results are shown in Fig. 6c. Both *ins* and *del* operations' performance becomes better when the number of thread blocks is increased since more concurrency can be obtained. However, the benefit from concurrency is restricted when the thread block number keeps increasing since more thread blocks also mean more contention on the heap nodes.

6.3 Performance with Synthetic Data

We evaluate the performance of inserting 1M, 8M, 64M keys into an empty priority queue and then deleting all keys from the priority queue while in both stages, it is insertion-only or deletion-only. We also consider different types of input keys, which are (1) uniformly chosen at random among 30-bit sized keys[3] (2) sorting the random keys in ascending order and (3) sorting the random keys in descending order. The collected results are shown in Table 2 [4].

Compared to P-Sync, BGPQ has an average 9.3X speedup. P-Sync's bottleneck is its insert operation since it only supports pipeline parallelism for operations at different levels of the heap. On the contrary, BGPQ explores the inter-node parallelism for both insertion and deletion. It shows that the support of fully concurrent insertion and DeleteMin operations has more potential in exploiting the parallelism than the strictly synchronized pipelined method.

BGPQ is much better than all CPU parallel baselines. It has 53X, 66X, and 83X speedup compared to TBB with 1M, 8M, and 64M keys respectively. When the number of keys being operated on the priority queue grows, BGPQ scales well and has a larger speedup. For SprayList, BGPQ achieves an average 10.8X speedup. As previous work showed[1], LJSL does not scale under large numbers of concurrent threads; thus, the performance is not competitive. SprayList's relaxed DeleteMin operation makes it faster than other algorithms. BGPQ beats CBPQ with around 21X speedup among all cases. CBPQ scales well with the data structure size since the most time-consuming part of CBPQ is the chunk splitting stage.

6.4 Performance under Different Heap Utilization

We measured each priority queue design [5] [6] under different utilization. Utilization is controlled by initializing the priority queue with different numbers of keys. In this experiment, each thread performs a pair of insert and DeleteMin operations, thereby preserving the utilization of the underlying data structure. We evaluate three different utilization levels: empty, 1M keys, and 8M keys, after which we execute 64M pairs of insert and DeleteMin operations.

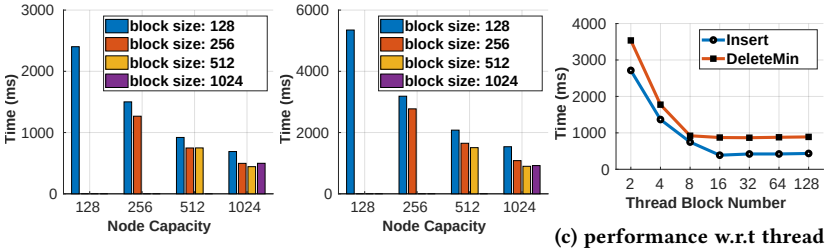
[2] Note that during the collaboration when a delete-heapify operation steals keys from an insert-heapify operation, the delete-heapify operation holds the lock of the root node and performs a spinlock until the insert-heapify operation fills the root node.

[3] Open sourced CBPQ implementation only supports 30-bit integer keys.

[4] Performance is in milli-second.

[5] P-Sync does not support concurrent insertion and retrieval operations.

[6] The CBPQ implementation restricts the number of chunks used. In our experiments, CBPQ runs out of chunks with 1e9 pre-allocated chunks.



(a) Insert operation (b) DeleteMin operation (c) performance w.r.t thread block numbers

Figure 6: BGPQ Performance w.r.t thread block sizes, node capacities and thread block numbers

			CPU Parallel PQs				GPU Parallel PQs		Improvement				
			TBB	Spraylist	CBPQ	LJSL	P-Sync	BGPQ	B/T	B/S	B/C	B/L	B/P
Ins & Del	Random	1M Keys	1,299	247	556	413	220	28	46.4	8.8	19.8	14.8	7.8
	Ascend		1,273	224	492	344	211	22	57.9	10.2	22.3	15.6	9.6
	Descend		1,151	212	480	315	197	21	54.8	10.2	22.8	15.0	9.4
	Random	8M Keys	12,043	1,822	4,021	8,284	1,517	212	56.8	8.6	19.0	39.1	7.2
	Ascend		11,291	1,613	4,017	5,649	1,515	173	65.3	9.3	23.2	32.7	8.8
	Descend		11,240	1,640	3,319	5,902	1,483	150	74.9	10.9	22.1	39.3	9.9
	Random	64M Keys	107,770	17,612	27,208	67,514	12,194	1,329	81.3	13.3	20.5	50.9	9.2
	Ascend		94,361	15,137	24,783	49,016	12,024	1,069	88.3	14.2	23.2	45.9	11.2
	Descend		93,850	14,859	22,877	44,911	11,892	1,193	78.7	12.5	19.2	37.6	10.0
Util.	Empty	64M Keys	53,320	149,803	N/A	51,305	N/A	1,452	36.7	103.2	N/A	35.3	N/A
	Init: 1M		57,074	12,721	N/A	52,088	N/A	1,441	39.6	8,8	N/A	36.1	N/A
	Init: 8M		72,473	12,967	N/A	53,806	N/A	1,487	48.7	8.7	N/A	36.2	N/A
0-1 KS	2^{200} node search tree		60,156	41,988	N/A	75,444	N/A	928	64.8	45.2	N/A	81.3	N/A
	2^{400} node search tree		2,453	1,484	N/A	3,504	N/A	27	90.9	55	N/A	129.8	N/A
	2^{600} node search tree		59,355	36,180	N/A	53,250	N/A	624	95.1	58	N/A	85.3	N/A
	2^{800} node search tree		5,207	2,850	N/A	5,256	N/A	52	100.1	54.8	N/A	101.1	N/A
	2^{1000} node search tree		156,670	95,476	N/A	156,249	N/A	1,807	86.7	52.8	N/A	86.5	N/A
A-star	5K*5K	10% obstacles	67,077	28,410	N/A	45,816	N/A	2,294	29.2	12.4	N/A	20.0	N/A
		20% obstacles	50,848	25,613	N/A	39,237	N/A	2,061	24.7	12.4	N/A	19.0	N/A
	10K*10K	10% obstacles	265,127	128,420	N/A	167,614	N/A	7,386	35.9	17.4	N/A	22.7	N/A
		20% obstacles	203,443	111,519	N/A	152,630	N/A	6,939	29.3	16.1	N/A	22.0	N/A
	20K*20K	10% obstacles	1,039,827	520,947	N/A	727,108	N/A	22,328	46.6	23.3	N/A	32.6	N/A
		20% obstacles	799,997	384,220	N/A	616,596	N/A	21,262	37.6	18.1	N/A	29.0	N/A

Table 2: Parallel Priority Queue Performance

As shown in Table 2, when the utilization increases, the performance of BGPQ maintains at the same level. Compared to CPU baselines, TBB suffers under high utilization, with a 36% slow down. SprayList performs worse when the initial data structure is empty. The significant performance degradation comes from the deleteMin operation, which has 20X more collisions. LJSL has only a 5% slowdown with high utilization but is still much slower than BGPQ.

Compared to SprayList, BGPQ achieves a maximum of 58X speedup for 600 items and a minimum 45X speedup for 200 items. Compared with TBB, the maximum speedup is 100X for 800 items, and the minimum speedup is 64X for 200 items. LJSL has similar performance with TBB.

6.5 Performance with Real-World Data

0-1 Knapsack We implement the branch and bound based 0-1 knapsack algorithm using different parallel priority queue implementations. In branch and bound algorithms, all visited nodes in the search tree are stored in the priority queue. Each time after the highest priority node is processed, its two branches in the search tree may be inserted into the heap, depending on if it is pruned by a bound condition. A thread block in BGPQ always retrieve a full node from the priority queue for load balancing purpose.

We use the generator [19] to generate large datasets with different numbers of items from 200 to 1000 with 2^{200} to 2^{1000} search space respectively. In Table 2, we show the performance of the knapsack application. [7].

[7]CBPQ's implementation only supports a 30-bit key thus is impossible to store a knapsack node that contains weight, profit, and level information. This is also why we don't evaluate CBPQ with A-star graph search.

A-star search for route planning A-star graph search is widely used in path finding and graph traversal algorithms. We focus on the A-star version that aims to find the shortest path between a given source and target node in a 2D grid with obstacles. We use the Manhattan distance as the admissible heuristic function. The priority function for any node v is the summation of the current distance from the source to v and the Manhattan distance from v to the target node. We use 3 different map sizes with 2 different obstacle rates. An obstacle rate r means r% of the nodes in the grid is an obstacle. The obstacles are randomly distributed in the grid, and there always exists a path from the start node to the target node. For any node in the grid, it has 8 directions to move.

We show the A-star performance in Table 2. We can observe that BGPQ based solutions are much faster than TBB, LJSL, and SprayList. The scaling trends show that with a larger grid, BGPQ can achieve higher speedup, and with an increased obstacle rate, the speedup reduces as the number of paths to explore reduces.

7 Related Work

Concurrent priority queues have been extensively studied in the literature [2, 5, 8, 9, 14, 21]. Nageshwara and Kuma used a fine-grained lock based heap implementation [21]. Hunt *et al.* improved it by combining bottom-up insertion with the LR-algorithm to reduce root contention [14]. Dragicevic and Bauer introduced a lock-free and linearizable heap that uses software transactional memory (STM) [9]. Deo and Prasad increases the node capacity of the heap to increase the parallelism [8]. However, their algorithm only supports pipeline parallelism.

Recent years have seen a surging number of skip-list based implementations [10, 13, 16, 24, 26]. Lotan and Shavit were the first to propose a quiescently consistent skip-list based priority queue[24]. Sundell and Tsigas introduced another skip-list based priority queue [27]. Linden and Jonsson [16] presented a skip-list based priority queue that batches physical deletions. Their design alleviated the contention on the head nodes. Alistarh *et al.* [1] introduced SprayList which is a relaxed algorithm that allows delete operations to randomly select a node within a certain priority range.

There are other data structures to implement concurrent priority queue ADT. Liu and Spear [17] introduced Mounds which supports fast insert and delete operations. Braginsky *et al.* reported a more involved design called CBPQ whcih consists of a series of linked chunks. Zhang and Dechev [30] presented a multi-dimensional linked list based priority queue (MD-List). Nodes in a MD-List contain multiple links to its child nodes arranged by their dimensionality which provides convenient concurrent accesses to different parts of the data structure.

All these concurrent priority queue algorithms are designed for multi-core CPUs. Our experiments has compared BGPQ with the state-of-the-art multi-core CPU implementations and demonstrated significant performance improvement.

8 Conclusion

This work presents BGPQ, a concurrent heap based priority queue implementation that is friendly to many-core GPUs. BGPQ exploits both intra-node and inter-node parallelism of the heap-based priority queue. We have proved the linearizability property of BGPQ. We implemented two search applications branch-and-bound knapsack and A-star algorithms with BGPQ. Experiments show a significant reduction in the execution time and memory footprint on GPUs, which demonstrates that there is significant potential for using heap for fine-grained scheduling on GPUs.

References

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 11–20.

[2] Rassul Ayani. 1990. Lr-algorithm: concurrent operations on priority queues. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*. IEEE, 22–25.

[3] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. 2016. CBPQ: High performance lock-free priority queue. In *European Conference on Parallel Processing*. Springer, 460–474.

[4] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. 2004. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Scandinavian Workshop on Algorithm Theory*. Springer, 480–492.

[5] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. 1998. A parallel priority queue with constant time operations. *J. Parallel and Distrib. Comput.* 49, 1 (1998), 4–21.

[6] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. 2014. The Adaptive Priority Queue with Elimination and Combining. In *Distributed Computing*, Fabian Kuhn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–420.

[7] Paolo G. Crosetto. 2019. CUPQ: a CUDA implementation of a Priority Queue applied to the many-to-many shortest path problem. https://doi.org/10.5281/zenodo.3595244

[8] Narsingh Deo and Sushil Prasad. 1992. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing* 6, 1 (1992), 87–98.

[9] Kristijan Dragicevic and Daniel Bauer. 2009. Optimization techniques for concurrent STM-based implementations: A concurrent binary heap as a case study. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[10] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[11] Oded Green, Robert McColl, and David A Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 331–340.

[12] Xi He, Dinesh Agarwal, and Sushil K Prasad. 2012. Design and implementation of a parallel priority queue on many-core architectures. In *High Performance Computing (HiPC), 2012 19th International Conference on*. IEEE, 1–10.

[13] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[14] Galen C Hunt, Maged M Michael, Srinivasan Parthasarathy, and Michael L Scott. 1996. An efficient algorithm for concurrent priority queue heaps. *Inform. Process. Lett.* 60, 3 (1996), 151–157.

[15] John Iacono, Ben Karsin, and Nodari Sitchinava. 2019. A parallel priority queue with fast updates for GPU architectures. *arXiv preprint arXiv:1908.09378* (2019).

[16] Jonatan Lindén and Bengt Jonsson. 2013. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*. Springer, 206–220.

[17] Yujie Liu and Michael Spear. 2012. A lock-free, array-based priority queue. *ACM SIGPLAN Notices* 47, 8 (2012), 323–324.

[18] Yujie Liu and Michael Spear. 2012. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP '12)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/ICPP.2012.42

[19] Silvano Martello, David Pisinger, and Paolo Toth. 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science* 45, 3 (1999), 414–424.

[20] N. Moscovici, N. Cohen, and E. Petrank. 2017. A GPU-Friendly Skiplist Algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 246–259. https://doi.org/10.1109/PACT.2017.13

[21] RV Nageshwara and Vipin Kumar. 1988. Concurrent access of priority queues. *IEEE Trans. Comput.* 37, 12 (1988), 1657–1665.

[22] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2009. Fast in-place sorting with cuda based on bitonic sort. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 403–410.

[23] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[24] Nir Shavit and Itay Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. IEEE, 263–268.

[25] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[26] Nir Shavit and Asaph Zemach. 1999. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, 113–122.

[27] Håkan Sundell and Philippas Tsigas. 2005. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005), 609–627.

[28] Orr Tamir, Adam Morrison, and Noam Rinetzky. 2016. A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 46)*, Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1–16. https://doi.org/10.4230/LIPIcs.OPODIS.2015.15

[29] Michael Voss, Rafael Asenjo, and James Reinders. 2019. *Pro TBB: C++ Parallel Programming with Threading Building Blocks* (1st ed.). Apress, USA.

[30] Deli Zhang and Damian Dechev. 2015. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2015), 613–626.

[31] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 369–380.