Accelerating Concurrent Heap on GPUs[1]

Team 15

SHUBHAM MITTAL

1 Introduction

Priority Queue is an abstract data type (ADT) where each element has a priority associated with it. An element with high priority is served before an element with low priority. Some of the well known applications of priority queue are: Dijkstra Algorithm, Huffman Encoding, Prim's Algorithm for finding MST of a graph, Discrete Event Simulation and many more.

Priority Queue is implemented as heap. This project involves accelerating this important ADT on GPU Architecture. The motivation for picking this project is - this ADT has not been extensively studied for its acceleration on GPUs due to the two major challenges which prevents from taking advantage of large amount of parallelism of GPU:

(1) Control Divergence
(2) Low memory locality

This project will focus on above two issues related to heap implementation on GPU and will try to resolve these for taking full advantage of the GPU parallelism. It will be shown that we can further enhance the performance of heap on GPU if we take into account the above two issues.

2 Literature review

Research work for parallelizing heap on CPUs includes the work by Rao and Kumar[5] which avoids taking the lock on the entire heap by a single global lock, instead associates each node with a lock, and makes insertion updates top-down so that the locking order of nodes prevents deadlock. Hunt et al. [4] adopts the same fine grained locking mechanism, but makes insertion updates bottom up while maintaining a top-down lock ordering. This implementation alleviates the contention at the root node. But since they were CPU implementations, control divergence of threads was not a problem and hence not tackled.

As pointed out above, parallelizing heap on GPU is not extensively studied and the only available work by He and others[3] is closely related to this project. This is the first parallel priority queue implementation on many-core architectures. It is based on the idea presented by Deo and Prasad[2] in 1992, which exploits the parallelism by increasing the node capacity in the heap to contain k keys (k >= 1). However, while it exploits intra-node parallelism, inter-node parallelism is not well exploited. It divides the heap into even and odd levels and uses barrier synchronization to make sure operations on two types of levels are never processed at the same time. Also, between every two consecutive barrier synchronization points, only one insertion/deletion request can be accepted, which severely limits the efficiency of its implementation on GPUs. No other work is available for heap on GPU as best of my knowledge.

Author's address: Shubham Mittal, shubham18101@iiitd.ac.in.

3 Milestones

S. No.	Milestone	Member
	Mid evaluation	
1	Implement Sequential Heap on CPU + Design and Analyse Concurrent Heap Algorithm for GPU in terms of Code implementation	Shubham
2	Implement Concurrent Insert-Key Operation on GPU	Shubham
	Final evaluation	
3	Implement Concurrent Delete-Key Operation on GPU	Shubham
4	Perform Fine Tuning and Optimisations, Stress Test, Performance Evaluation and Measurements	Shubham

4 Approach and Analysis

Generalised Heap(Fig. 1) is an extension of common heap. In this project, min-heap will be referred to as heap. Every node in the generalised heap will contain **k**(some power of 2) keys, together called a *batch*, with the following properties:-

(1) Root node will contain smallest k elements in sorted order.
(2) Every batch within itself is sorted.
(3) Smallest key in any batch ≥ largest key in its parent batch except root node since it has no parent.

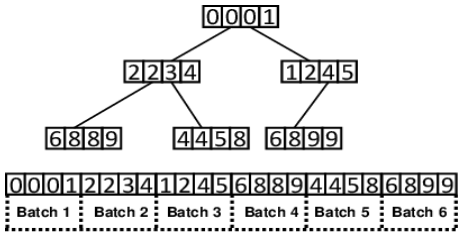

Fig. 1. A generalized heap

This is done so that we will be having less control divergence and and maximum memory coalescing since a batch of k keys will move along the same path during any operation of the heap. We will be having a *partial buffer* to avoid insertion of less than k keys in heap and keep them on hold till the time total keys becomes k. Partial Buffer will have size $k - 1$ so that whenever it overflows, we come to know that an insert operation is required in heap.

Insertion: Let's we have to insert *insertKeys* in our heap. We will first sort *insertKeys* using **Bitonic Sort** which is highly parallel in nature and hence perfect for GPU Architecture. We will then propagate from the root node to the target node while merging with the nodes present in path from root to target using **MergeAndSort** such that it always satisfies the root properties.(Fig. 2)

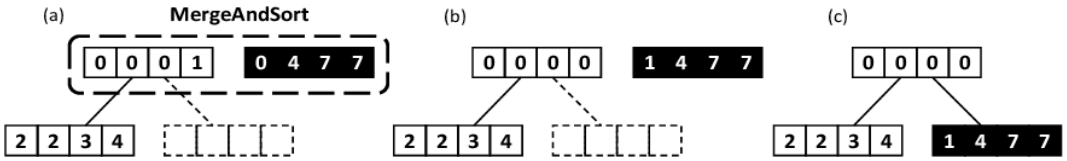

Fig. 2. Insertion in generalised heap

Deletion: We will delete exactly k nodes from the root of the heap and copy the last node into the root node and perform the heapify operation until we satisfy all the generalised-heap properties described above.(Fig. 3)

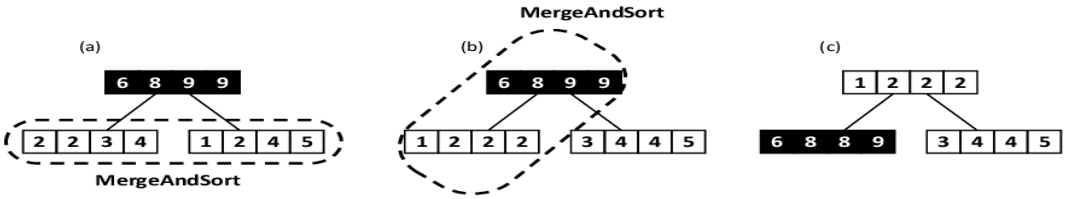

Fig. 3. Deletion in generalised heap

For the **MergeAndSort**, two pointers approach is not parallisable, so I used binary search to find right index of an element in the final array. On comparing binary search approach with two pointers approach, former was found to be performing much better.

5 Challenges

(1) Performing both the operations concurrently requires lock for each node in the heap. Ensuring no deadlock was one of the main challenging task. To prevent deadlock, lock ordering was used such that to take the lock in child node, thread must have lock of parent node.

(2) Ensuring that the kernel which is invoked first should take the lock on root node first. Since if we launch kernels in multiple streams then any of them can execute first which may give wrong output. The idea used is to give a unique ID to each kernel invocation and use atomic operation to match it will global ID inside heap.(Fig. 4)

```
while(atomicCAS(&(heap -> global_id), my_id, 0) != my_id);
take_lock(&heap_locks[ROOT_NODE_IDX], AVAILABLE, INUSE);
heap -> global_id = my_id + 1;
```

Fig. 4. Taking lock on root node

(3) For *MergeAndSort*, a number of methods were tried and finally binary search for each index is found to be best but implementing it on the GPU was a tedious task due to arbitrary memory access. To reduce global memory access, all data was copied into shared memory and lots of synchronisation points were required to do so. Reducing the number of **__syncthreads()** is one of motive and challenging at the same time to improve the performance.

6 Results

Heap sort was used as a benchmark. First all keys were inserted and then deleted . This will sort all inserted keys. All measurements are in milliseconds.

Batch Size = 1024 | Number of CudaStreams = 16					
Number of keys	GPU kernel (in ms)	Including Memcpy (in ms)	Sequential Heap (in ms)	Speedup (kernels)	Speedup (Memcpy)
~2^{22}	65.729	164.279	2017.25	30.69	12.27
~2^{23}	132.192	324.97	4633.37	35.05	14.25
~2^{24}	257.641	414.064	10976.4	42.60	26.50
~2^{25}	506.131	686.576	25105.5	49.60	36.56
~2^{26}	1000.94	1245.47	59183.6	59.12	47.51
~2^{27}	1996.96	2417.35	133369	66.78	55.17

Table 1. GPU results compared with my CPU implementation of heap

Batch Size = 1024 | Number of CudaStreams = 16					
Number of keys	GPU kernel (in ms)	Including Memcpy (in ms)	STL Heap (in ms)	Speedup (kernels)	Speedup (Memcpy)
~2^{22}	65.729	164.279	4540.38	69.07	27.63
~2^{23}	132.192	324.97	10131.3	76.64	31.17
~2^{24}	257.641	414.064	22563.9	87.57	54.49
~2^{25}	506.131	686.576	50848.7	100.46	74.06
~2^{26}	1000.94	1245.47	117353	117.24	94.22
~2^{27}	1996.96	2417.35	262323	131.36	108.51

Table 2. GPU results compared with STL heap

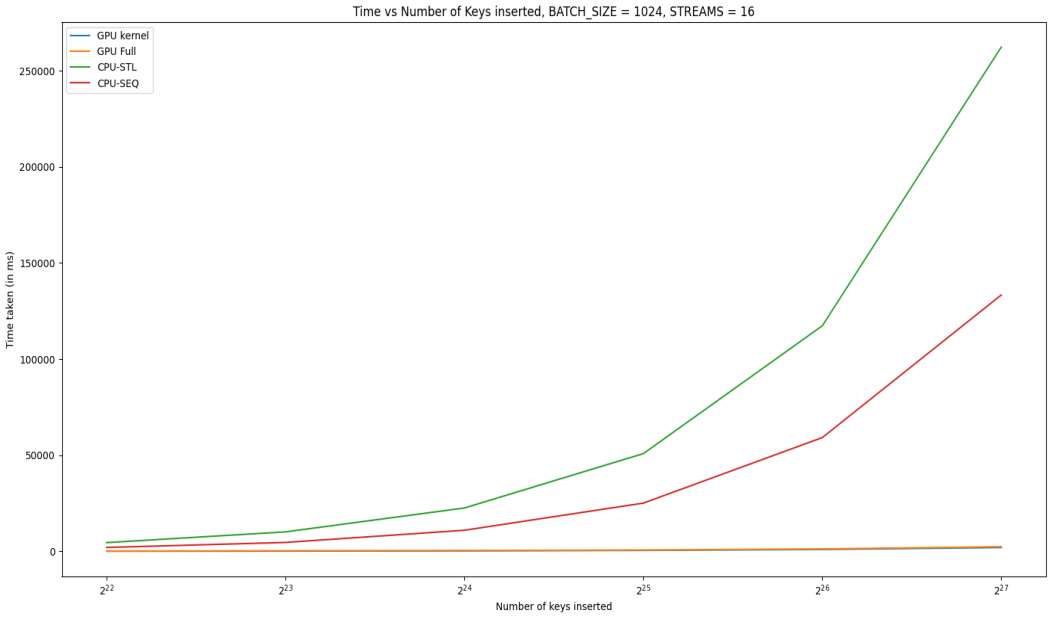

Fig. 5. From Table 1 and 2

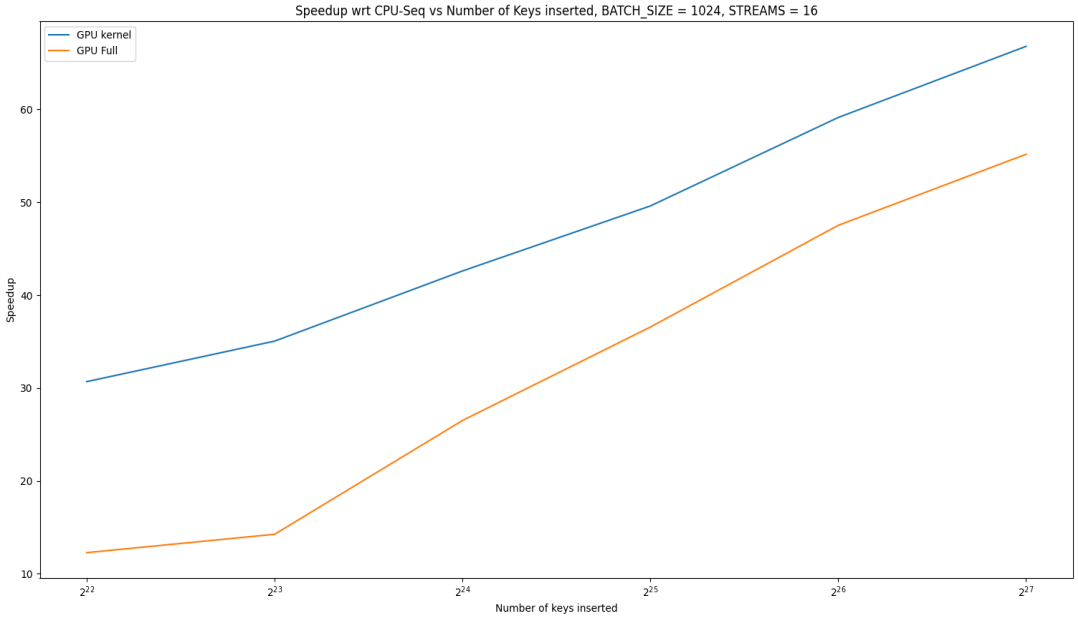

Fig. 6. From Table 1

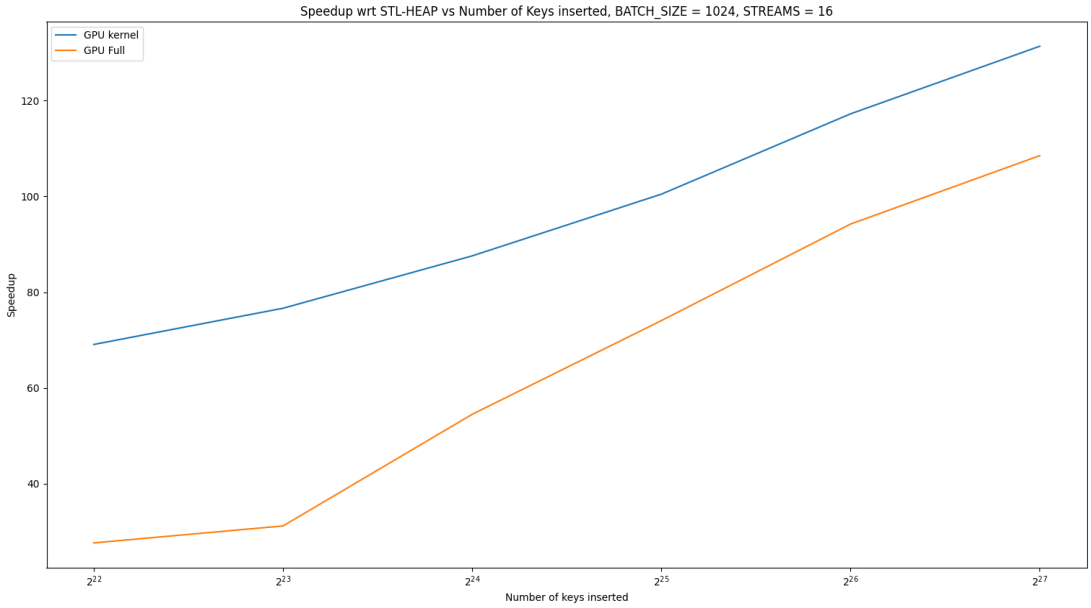

Fig. 7. From Table 2

As we expected, our GPU implementation of Heap is outperforming CPU implementation by a large number. Also, as can be seen in Fig 5, there is not much overhead for memory transfer and data structure initialization. Since each block can have at most 1024 threads, this limits the value of batch size to be 1024 at-max. STL heap is found to be performing worst than my own implementation of heap which is not expected. Searching the Web, it was found that STL heap is slow because it provides more features than just insert and delete and to provide those functionalities, it has to maintain certain data structures inside it which lead to extra overhead and hence perform worst than a naive heap implementation as shown by results.

7 Conclusion

This is a nice algorithm for Concurrent Heap on GPUs but it is suitable only for bulk operations like inserting 100M keys together or deleting but not suitable for algorithms involving insertion or deletion of single element like in case of Prim's Algorithm for finding minimum spanning tree of a graph, it will perform very badly.

Also, though concurrent, only three operations were getting executed concurrently at any point of time which is attributed to following facts:

(1) Only one operation can perform on any node and since root is node is required for every operation, there is sequential order in taking root node lock.
(2) For my implementation on this particular GPU, since depth of my tree was not greater than ˜17, the time when 4th operation takes lock on root node, first operation has already finished its execution and gets exit attributed to the high performance of algorithm, speedy GPU and low tree depth. Thereby, ˜3 operations were running concurrenly though we expect more.

If it is possible to reduce contention over top-level nodes somehow, more operations would be able to run concurrently and we can see further speedups. As correct to my knowledge and authors too, this is the best

algorithm as of now for accelerating heap on GPU and the second attempt made in 2019 by [1] to improve heap on GPU and first attempt was made in 2012 by [3].

References

[1] Yan-Hao Chen, Fei Hua, C. Huang, Jeremy Bierema, Chi Zhang, and E. Z. Zhang. 2019. Accelerating Concurrent Heap on GPUs. *ArXiv* abs/1906.06504 (2019). https://arxiv.org/abs/1906.06504

[2] Narsingh Deo and Sushil Prasad. 1992. Parallel Heap: An Optimal Parallel Priority Queue. *J. Supercomput.* 6, 1 (March 1992), 87–98. https://doi.org/10.1007/BF00128644

[3] X. He, D. Agarwal, and S. K. Prasad. 2012. Design and implementation of a parallel priority queue on many-core architectures. In *2012 19th International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HiPC.2012.6507490

[4] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. 1996. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Inf. Process. Lett.* 60, 3 (Nov. 1996), 151–157. https://doi.org/10.1016/S0020-0190(96)00148-2

[5] RV Nageshwara and Vipin Kumar. 1988. Concurrent access of priority queues. *IEEE Trans. Comput.* 37, Dec (1988), 1657–1665. https://doi.org/10.1109/12.9744