**© 2019**

**Chaozhang Huang**

**ALL RIGHTS RESERVED**

**A MODEL FOR CONCURRENT PRIORITY QUEUE ON MANY-CORE ARCHITECTURES**

**By**

**CHAOZHANG HUANG**

**A thesis submitted to the**

**School of Graduate Studies**

**Rutgers, The State University of New Jersey**

**In partial fulfillment of the requirements**

**For the degree of**

**Master of Science**

**Graduate Program in Computer Science**

**Written under the direction of**

**Zheng Zhang**

**And approved by**

_____

_____

_____

**New Brunswick, New Jersey**

**OCTOBER, 2019**

**ABSTRACT OF THE THESIS**

**A Model for Concurrent Priority Queue on Many-core Architectures**

**By CHAOZHANG HUANG**

**Thesis Director:**

**Zheng Zhang**

The concurrent priority queue is one of the shared memory data structures that can be dynamically maintained and updated for communication among co-running threads, which allows data with high priority to be served before others in applications that require complex asynchronous communication patterns. For instance, the following algorithms can take advantages of a concurrent priority queue: Dijkstras shortest path algorithm, Prims minimal spanning tree algorithm, data compression and A* search in artificial intelligence, which are building blocks for complex system software including operating system scheduler, interruption handler, and garbage collection. However, it is challenging to exploit the performance of heap-based priority queue on many-core architectures like GPUs due to control divergence and memory irregularity. This thesis focuses on the development and evaluation of a model for concurrent priority queue on GPU. We present an efficient model for heap-based concurrent priority queue, called the generalized heap, which exploits both inter-node and intra-node parallelism by incorporating with multi-state locking mechanism and batch processing on data nodes. The performance of the concurrent priority queue based on the generalized heap model is thoroughly evaluated against previous serial and GPU implementations. We show a maximum of 19.49X and 2.11X speedup compared to previous serial and GPU implementation, respectively. We also apply our generalized heap model in real-world applications such as single-source shortest path and 0/1 knapsack problem with up to 1.23X and 12.19X speedup, respectively.

# Acknowledgements

Throughout my study and the writing of this thesis, I have received enormous support and assistance.

Foremost, I would like to express my deepest appreciation to my advisor Prof. Zheng Zhang for her guidance and unwavering support throughout my study and research at Rutgers University.

In addition, I would also like to extend my deepest gratitude to my parents, without whose continuous support my study would not be possible.

I must also thank members of my committee: Prof. Desheng Zhang and Prof. Yongfeng Zhang. Their help and advice are indispensable for the completion of this thesis.

Moreover, I am grateful for the help and motivation I received from my lab members: Yanhao Chen and Fei Hua. Their companion and help are crucial to my research experience.

# Table of Contents

# Chapter 1

# Introduction

Not all data is created equal. Based on the application, it is often time worthy to process part of the data before others to avoid redundant and unnecessary computation. For example, applications that are developed based on this idea include Dijkstra's shortest path algorithm[1], Prim's minimum spanning tree algorithm[2], A* searching [3] and various branch-and-bound based algorithm that solves different combinatorial optimization problems. Proper prioritization of data allowed these applications to vastly prune the search space and delivers better performance.

This situation emerges even more often with the recent development of parallel computing on many-core accelerators. For example, Nvidia GPUs are shown to achieve substantially better throughput compared to Intel CPUs [4]. However, efficient concurrent data structures are needed to integrate the raw throughput of hardware with parallel algorithms. A concurrent data structure is a shared memory data structure that can be dynamically maintained and updated for communication among co-running threads.

Traditionally, the priority of data can be utilized by maintaining data with a priority queue. A priority queue is an abstract data type that functionally behaves like a regular queue or stack structures, allowing insert and delete operations, but with the difference being each element is associated with a priority. In a priority queue, an element with high priority is always served before an element with low priority. The priority queue is often time implemented using the *heap*, a tree-based abstract data structure that organizes elements in tree nodes based on its priority. Though previous research on parallel priority queues [5][6][7][8] are available, their implementation suffers from various limitations and bottlenecks as described in section 2.3.

This thesis focuses on the development and evaluation of a model for concurrent priority queue for many-core architectures. We present an efficient model for concurrent priority queue that is well-suited for many-core general purpose accelerators, the concurrent *generalized heap*.

Though the concurrent *generalized heap* is implemented and evaluated on GPU, it can be applied to other many-core architectures with vector processing unit as well. It is an efficient concurrent implementation of the *heap* data structure. The concurrent *generalized heap* takes advantage of the model presented by Deo and Prasaed [8], allowing multiple elements to be stored in single tree node with two fundamental differences. Assume the node capacity is $k$, the concurrent *generalized heap* provides partial operations while [8] only allowed operations with exactly $k$ elements. Moreover, while [8] only focused on intra-node parallelism, the concurrent *generalized heap* exploits both intra-node parallelism and inter-node parallelism. The design and implementation of the concurrent *generalized heap* will be discussed in Chapter 3 and 4, respectively. We also perform a comprehensive evaluation of the concurrent *generalized heap*. Signifying the relationship between heap performance and various parameters including: node capacity, percentage of partial operations, number of concurrent thread blocks, initial heap utilization. Experiments also show that the concurrent *generalized heap* has up to 19.49X and 2.11X speedup compared to sequential CPU and existing GPU implementation. With simple integration, the concurrent *generalized heap* can improve the parallel SSSP (Single Source Shortest Path) algorithm by up to 1.23X and the 0/1 knapsack by 12.19X. This thesis shed lights on potential to integrate concurrent data structures with real-world applications to achieve substantial speedup and exploit parallelism for parallel computing algorithms on many-core accelerators.

# Chapter 2

# Background

## 2.1 Priority Queue

Traditionally, to achieve the prioritization of data objects, we store and retrieve data using a *priority queue*, which is a queue-like abstract data type that assigns priority to each data element. In contrast to a *queue*, which serves data in a first-in, first-out manner, a priority queue, on the other hand, has a special property that a data element with higher priority is always served before an element with lower priority.

### 2.1.1 Heap Data Structure

In practice, we often implement priority queue and store data elements using the *heap*, a fundamental tree-based data structure that treat key value of each node as its priority. The resulting tree of heap and must satisfy the *heap property*: if a node $P$ is the parent of some node $C$, then the key value of $P$ must be greater or equal to (in the case of a max-heap) or less than or equal to (in the case of a min-heap) the key value of $C$. The tree representation of a heap is often a complete binary tree, *i.e.* it is always filled except the last level. An example of a min-heap can be found in Figure 2.1.

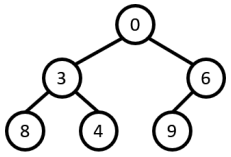
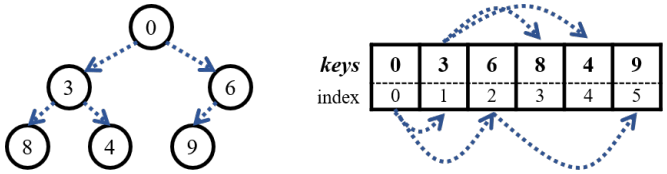



Figure 2.1: An example of min-heap where numbers are keys of tree nodes

### 2.1.2 Array Representation

The most space-efficient way to store a heap is to treat it as an array instead of a linked list since it not only neglect the space required to store various pointers but also provides O(1) complexity for accessing any node, whereas with a linked list one must iterate from the root every time to get to other nodes. The array representation of heap works as follows:

1. The root node will be stored at Arr[0], the first item in the array.
2. Suppose the total number of nodes is $N$, then for $0 \leq\ i \leq\ N$, the parent and child nodes of the $i^{th}$ node can be obtained through:
   - Parent($i$) = Arr[$(i - 1) \div 2$]
   - Left($i$) = Arr[$(2 \times\ i) + 1$]
   - Right($i$) = Arr[$(2 \times\ i) + 2$]




**(a)** Tree representation **(b)** Array representation

Figure 2.2: Array representation of a min-heap

An example of the array representation of the min-heap presented in Fig. 2.1 is shown in Fig. 2.2. Note that the dashed line in Fig. 2.2 (b) only resembles the tree edges in Fig. 2.2 (a) while they are not physically stored in the memory.

### 2.1.3 Basic Operations

The heap has two fundamental operations: insertion update and deletion update. The insertion operation inserts a new node to the correct location in the tree such that the heap property is not violated. Similarly, the delete operation retrieves the node with either the smallest key (in the case of a min-heap) or the largest key (in the case of a max-heap) and updates the tree accordingly. Both operations take logarithmic time.

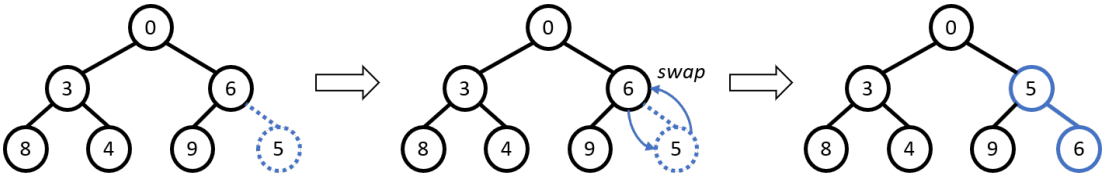



Figure 2.3: Bottom-up Insertion Update, with *key* = 5 to be inserted

**Insertion Update**

An insertion update inserts a node with a new key-value to the correct location in the tree through repeated propagation. Using the min-heap presented in Fig. 2.1 as an example, an insertion update can be performed by repeatedly applying the following steps: first, the new key node is appended to the last position in the tree. Then, compare the key value of the new node with its parent's key value, if the new key is smaller than its parent's key value, swap this node with its parent, otherwise, this node has reached the correct location and we can terminate. An example of an insertion update using the above *bottom-up* insertion algorithm is shown in Fig. 2.3 with a node with *key* = 5 is to be inserted. We show the *bottom-up* insertion algorithm discussed above in Alg. 1.

**Algorithm 1:** Insertion Update

```
1 Procedure insert: (key)
2 last_node = key
3 n = last_node
4 while n != root do
5     if n < Parent(n) then
6         Swap(n, Parent(n))
7     else
8         break
```

**Deletion Update**

A deletion update retrieves the key at the root node and updates the heap to preserve the heap property. Using min-heap as an example, the deletion procedure consists of three steps: (1) it removes the current root node, which contains the minimum key, from the heap. (2) it moves the node to the root position and starts a *heapify* process from the new root: it repeatedly compares and swaps with its child with the smallest key until the heap property is preserved.

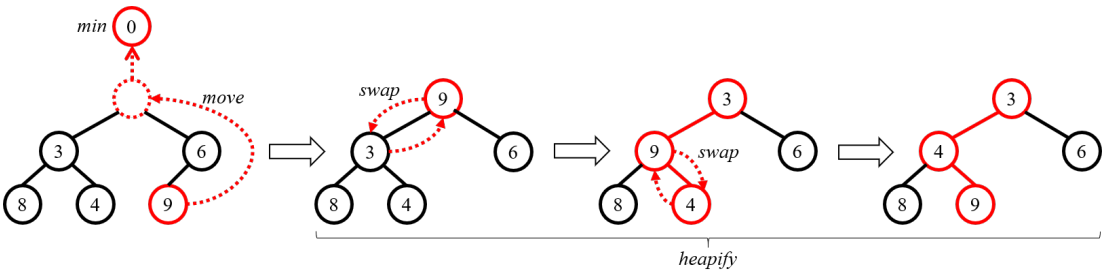



Figure 2.4: Deletion Update

(3) When the heap property is again preserved, we return the minimum key obtained from step (1) and terminate. An example of the deletion update for min-heap is shown in Fig. 2.4. The corresponding algorithm for deletion update is shown in Alg. 2

**Algorithm 2:** Deletion

```
1  Procedure delete: ()
2  min = root, n = root = last_node
3  last_node = null
4  while Left(n) != null or Right(n) != null do
5      c = min(Left(n), Right(n))
6      if n < c then
7          break
8      else
9          Swap(n, c)
10 return min
```

## 2.2 GPU: Modern Many-core Architecture

GPU (Graph Processing Unit) is an example of the modern multi-core accelerator. The differences in architecture between CPU (Central Processing Unit) and GPU is shown in Fig. 2.5 [9]. Traditionally, a CPU is consist of several ALUs (Arithmetic Logic Units) that performs arithmetic and bit-wise operations, a control unit corresponding to the ALUs, fast but small cache memories and a large DRAM (Dynamic Random Access Memory). In contrast, a GPU might consist of hundreds or thousands of processing units and corresponding control units. The massive amount of physical cores provided by the GPU allows one to execute tasks in parallel to achieve potential significant speedup. Fig. 2.6 [9] shows the performance advantages on Nvidia GPUs over the years versus Intel CPUs where the X-axis corresponds to different years and the

Y-axis shows the theoretical throughput of GPU architecture at that year.


Figure 2.5: Architectural Difference between CPU and GPU

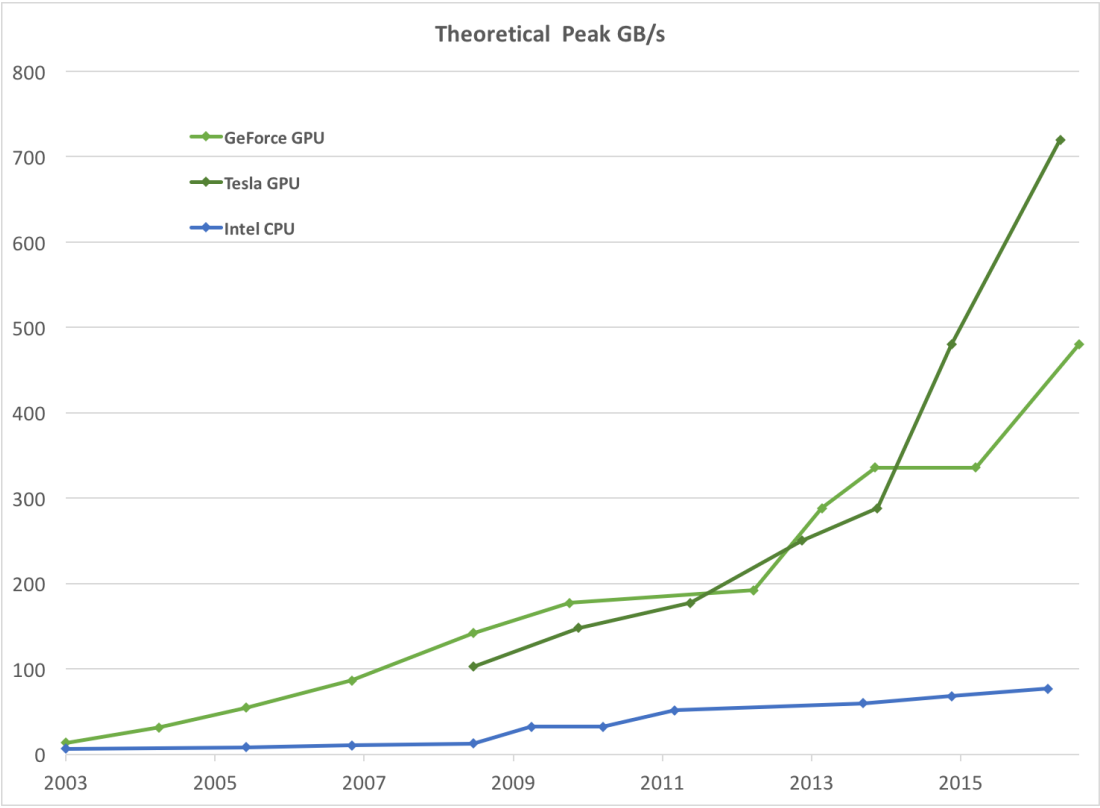



Figure 2.6: Performance Difference between CPU and GPU

### 2.2.1 The SIMT Model

Nvidia GPUs adopted *SIMT* (Single Instruction Multiple Thread) as the parallel execution model [10][11]. In the SIMT model of Nvidia GPU, the basic unit of execution is a thread *warp*, which is a collection of 32 threads (in the current implementation). Threads in the same

warp are executed simultaneously on an SM (Streaming Multiprocessor), which is a scalable array of execution units that shares the same set of registers and shared memory. A thread *block* is consists of multiple thread warps. When a GPU program starts, the host CPU invokes a kernel *grid*, which contains a collection of thread blocks. The thread blocks then are distributed to different SMs for execution. Threads in the same thread block are executed concurrently on one SM and multiple blocks can be executed concurrently on one SM as well. A demonstration of the SIMT model can be found in Figure 2.7.

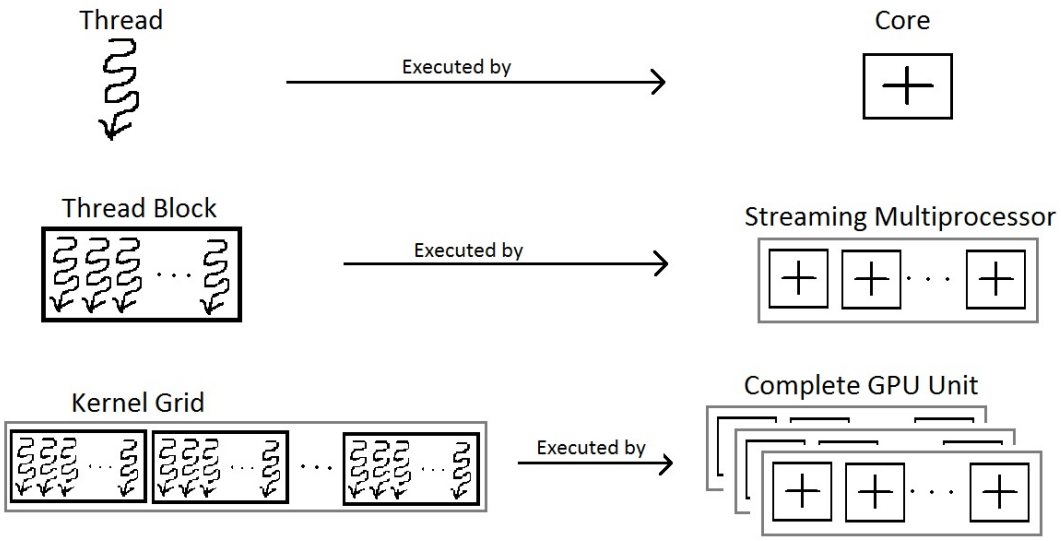



Figure 2.7: The SIMT Model on Nvidia GPU [12]

The SIMT model provides a limited set of built-in synchronization primitives. For example, invocation between kernels are serialized, which can be used as an implicit synchronization among all threads. Also, barrier synchronization is provided to synchronize threads within the same thread block. Although no built-in locking intrinsic is provided, workarounds that utilize the built-in atomic CAS (compare and warp) operation to implement synchronization mechanics exist.

As the name *SIMT* (Single Instruction Multiple Thread) implies, instructions on GPU are executed by multiple concurrent threads in a lock-step manner. Threads belong to the same warp can only execute one instruction at a time, which means if threads in a warp need to process different instructions at the same time, these instructions will be serialized. This phenomenon is called *control divergence*. An example of the potential hazard caused by control

divergence can be found in Figure 2.8 where T1 ... T4 are active threads within the same warp. Figure 2.8(a) describes a situation when divergence does not exist, all threads within the warp are assigned the same instructions, the utilization of threads can be maximized. However, if threads are assigned different instructions, then the instructions will be serialized, cutting the throughput of the application by half in the example showed in Figure 2.8(b). In practice, the performance hazard caused by serious control divergence could be a lot worse, which makes *control divergence* an important factor to be considered when implementing GPU programs.

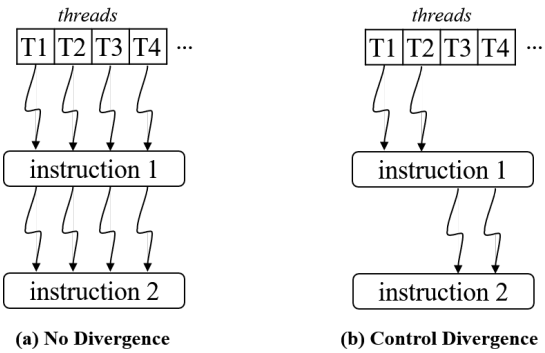



Figure 2.8: Control Divergence on GPU

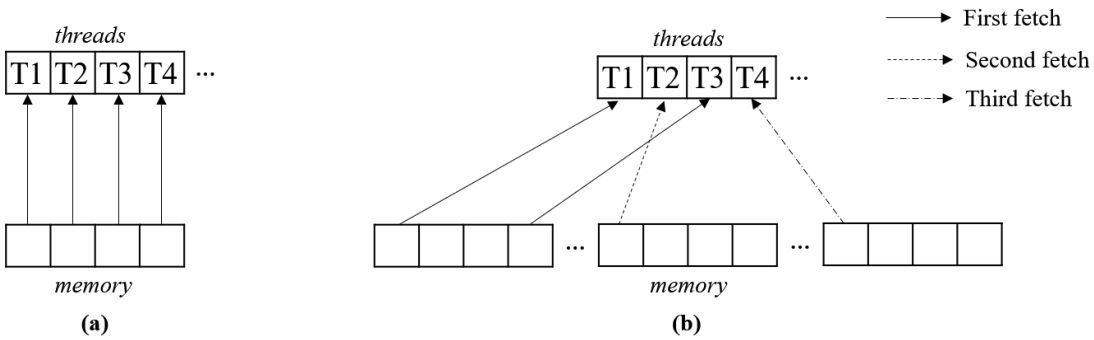



Figure 2.9: Optimized v.s. Poor Memory Locality

Another factor that needs to be considered when implementing GPU programs is *memory locality*. As aforementioned, the basic execution unit on GPU is a thread warp. The execution of a warp cannot be started unless all data required is fetched. However, the physical memory is organized in contiguous blocks and data is fetched block by block. If threads in the same warp require non-contiguous data access, then multiple data fetches need to be done, which results in a poor memory locality and hinders the overall performance of the execution. On the

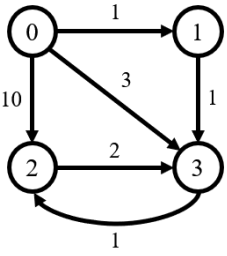
other hand, memory parallelism can be exploited by letting threads in the same warp to process contiguous data, which minimized the number of memory fetches needed and thus maximizes the performance. A comparison between optimized and poor *memory locality* on GPU can be found in Figure 2.9.

### 2.2.2 Real-World Applications

Modern many-core architectures can be used to accelerate various real-world applications. To unlock the full potential of parallel computing on such architectures, efficient implementation of concurrent data structure is needed. This thesis propose a concurrent heap model that can be applied to speedup real-world parallel computation. In this thesis, it is integrated and evaluated with two real-world applications: the SSSP (Single Source Shortest Path) algorithm and the branch and bound algorithm for the 0/1 knapsack problem. While the details of the concurrent heap model and the result of the evaluation will be discussed in Section 3.3 and Section 5.7, respectively. The two applications we integrated with will be briefly introduced in the following section.

#### The SSSP (Single Source Shortest Path) Problem

The SSSP (Single Source Shortest Path) problem defined as a process to find the shortest distance between the source node and all other nodes in the graph given weights of all edges in a graph. An example of the SSSP problem is shown in Figure 2.10. The program starts with a distance array initialized to infinity. Through processing edges iterative, the distance array gradually converges to the solution.




| # Iterations | Distances | | |
|---|---|---|---|
| | Node 1 | Node 2 | Node 3 |
| 0 | ∞ | ∞ | ∞ |
| 1 | 1 | 10 | 3 |
| 2 | 1 | 10 | 2 |
| 3 | 1 | 3 | 2 |

Figure 2.10: Sample SSSP Problem

The Bellman-Ford algorithm is widely used and studied to solve the SSSP problem since it

can be easily parallelized. The pseudo-code of the sequential Bellman-Ford algorithm is shown in Algorithm 3. For example, the edge relaxing operation (line 6 to 8) can be easily parallelized such that each thread is assigned to process a portion of the edges.

**Algorithm 3:** Sequential Bellman-Ford Algorithm

```
1 Procedure SSSP: (distance[], vertices, edges)
2 for each vertex v in vertices do
3     distance[v] = inf
4 distance[source] = 0
5 for i from 1 to size(vertices) - 1 do
6     for each edge (u, v) with weight w in edges do
7         if distance[u] + w < distance[v] then
8             distance[v] = distance[u] + w
9 return distance[]
```

Previous research has been done on concurrent implementation of the SSSP problem. Gunrock [13], a well known parallel iterative graph processing library on GPUs, implements a compute-advance model to solve iterative graph processing problems like the SSSP. In the compute-advance model, a node will be marked as an *active* node if a node is updated during the current iteration. Then, only edges of active nodes will be processed concurrently in the next iteration since edges of inactive nodes will not affect the distance[] array anyway.

**Incorporation with Heap**: For an edge $(u, v)$ with weight $w$, assuming greedily that the shortest destination distance[$v$] always comes from a smaller source distance[$u$] regardless of the weight of the edge, then once we processed edges from *active* nodes with smaller distance, the rest of the edges connects to the same destination node can be skipped. Using current distance as key values, active nodes of current iteration can be stored to the heap and be deleted from the heap for processing in the next iteration. In this way, edges of active nodes with smaller current distance will be processed before other edges in the next iteration, which can avoid unnecessary updates to the distance[] array. As a result, the overhead of making intermediate updates and the number of nodes the algorithm needs to explore can be reduced, resulting in a better performance. Details evaluation of solving SSSP with the concurrent *generalized* heap will be presented in Section 5.7.1.

**The Branch and Bound Algorithm for the 0/1 Knapsack Problem**

Many real-world decision-making problems can be broke down to the knapsack problem. The knapsack problem is defined as: given a knapsack with weight capacity $W$, weights and profits for some items, find the maximum profit we can obtain by putting different combinations of items in the knapsack while managing not to exceed to weight capacity. The 0/1 knapsack problem is a category of knapsack problem where if an item were to be selected, it must be selected in its entirety, no partial selection is allowed.

The branch and bound algorithm is one of the algorithms to solve the knapsack problem. It models the knapsack problem as finding a path from the root node to a leave node in the decision tree where the $i^{th}$ level of the tree denotes whether or not the $i^{th}$ item is selected. An example of the Knapsack problem can be found in Figure 2.11. The sequential implementation of the branch and bound algorithm first sort the items by the decreasing order of weight/profit ratio. Then it traverses the decision tree starting at the root node and keeps track of global maximum profit. At each node, situations of either select the item at the current level or not will be considered, the corresponding cumulative weight and profit of the specific path from the root to this node will be updated. A *profit bound* of the current node will also be computed. If the profit bound of the current node plus the cumulative profit is less than the global maximum or the cumulative weight exceeds weight capacity, simply discard this path. Otherwise, the current node is enqueued into a priority queue with cumulative profit as the key value. The traversal will continue at the node that has the largest priority until the queue becomes empty. The pseudo-code of the sequential branch and bound algorithm can be found in Algorithm 4. Note that the priority queue can be implemented using the heap.

The branch and bound algorithm is also straight forward to take advantage of parallel computing. The node exploration process can be parallelized using the concurrent *generalized heap* model introduced in Section 3.3. Optimization can be integrated into the parallel version of the branch and bound knapsack algorithm. Details of the integration and evaluation of the parallel branch and bound knapsack algorithm with concurrent *generalized heap* will be discussed in Section 5.7.2.

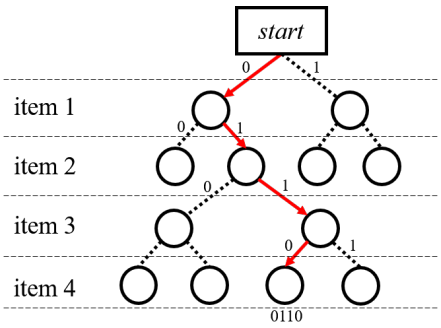



Figure 2.11: Sample 0/1 Knapsack problem with 0110 as the solution (items 2 and 3 are selected).

## 2.3 Previous Work on Concurrent Heap Models

An intuitive idea to implement the parallel heap on a multi-core architecture is to lock the entire heap while one thread is modifying it. This idea, though straight forward, is unfeasible since that it only allows one access to the heap at a time, which is a waste to the massive paralleling capability of modern multi-core architecture. Previous work by Nageshwara *et al*. [5] in 1988 made the parallel heap on CPU possible by applying the locking mechanism directly on the node level while using a top-down insertion method to avoid the possible deadlock. However, their locking mechanism did not solve the problem of contention on the root node. Later in 1996, Hunt *et al*. [6] further extended this locking mechanism by implementing a bottom-up insertion method while keeping the top down locking order.

He *et al*. [7] proposed a GPU heap model based on ideas presented by [8] in 1992. It exploits intra-node parallelism by increasing the node capacity in the heap where each node can store $k \geq 1$ keys. However, their implementation has limited support for inter-node parallelism. It divides the heap into odd and even levels by barrier synchronization and allowing only one operation to be processed between two consecutive levels, which severely limits the efficiency of this implementation on GPU.

**Algorithm 4:** Sequential Branch and Bound Algorithm for Knapsack

```
1  Procedure knapsack: (profits, weights, w)
2  sort(profits, weights)
3  pq = new heap; pq.push(root), max_profit = 0
4  while pq not empty do
5      u = pq.pop()
6      if u is last item then return
7      v = node(); v.level = u.level + 1
8      v.weight = u.weight + weights[v.level]
9      v.profit = u.profit + profits[v.level]
10     if v.weight < w and v.profit > max_profit then
11         max_profit = v.profit
12     v.bound = compute_bound(v)
13     // select the item
14     if v.bound + v.profit > max_profit then
15         pq.push(v)
16     // do not select the item
17     v.weight = u.weight; v.profit = u.profit; v.bound = compute_bound(v)
18     if v.bound + v.profit > max_profit then
19         pq.push(v)
20 return max_profit
```

# Chapter 3

# Methodology

Conventional heap structure is inherently hard to take advantage of parallelism provided by GPUs since contentions between parent and child nodes exist, which hinders the overall performance of the heap. While existing work tried to tackle this issue, their solutions still suffer from potential performance hazards such as *control divergence* and low *memory locality*. To fully exploit the possible parallelism of heap operations, we propose a modified heap model that supports concurrent insertion and deletion updates where each node in the tree is consists of a batch of key values instead of only one in the traditional model. The proposed model is adapted to exploit the parallelism of current many-core architectures like GPUs by enabling inter-node and intra-node parallelism with minimal control divergence. In the following section, we refer to the proposed heap model as the *generalized heap* and discuss its structure, sequential operations and concurrent operations based on a min-heap implementation. The max-heap version of the *generalized heap* can be easily realized by negating a min-heap.

## 3.1 The Generalized Heap

Instead of storing a binary tree of key values in the conventional heap, the *generalized heap* model consists of a binary tree of data *batches*. Each tree node is a data *batch*[1] that contains exactly $k$[2] key values. While heap operations do not guarantee the number of items in the tree is always equal to multiple of $k$, an extra data *batch* is used to handle these lingering items. This extra data *batch* is called the *partial buffer*. The *partial buffer* store up to $k-1$ key values that are larger than the key values in the root node, which ensures that the smallest keys (those

[1] Since a data batch is a tree node in the *generalized heap*, we will use the word "batch" and "node" interchangeably throughout the dissertation.

[2] $k$ is equal to the node capacity, which is the size of a data batch

with the highest priority) are always store at the root *batch*.

### 3.1.1 Generalized Heap Properties

Similar to the conventional *heap*, to correctly preserve the priority of the data, the resulting tree of the *generalized heap* must follow the *generalized heap properties* which are formally defined as follows:

**Property 1**. Given any node $n$ in the generalized heap, the keys in $n$ are sorted in ascending order:

$$\forall i \in [1, k) : node[c][i] \leq node[c][i+1]$$

**Property 2**. Given any node $n$ in the generalized heap and its parent node $p$, the smallest key in $n$ is always larger than or equal to the largest key in $p$:

$$\min_{i=1..k} node[n][i] \geq \max_{j=1..k} node[p][j]$$

**Property 3**. Given a partial buffer $b$ of size $s$, all the keys in $b$ are sorted in ascending order:

$$\forall i \in [1, s) : node[b][i] \leq node[b][i+1]$$

**Property 4**. Given a partial buffer $b$ of size $s$, all the keys in $b$ are larger than or equal to the largest key in the root node $r$:

$$\forall i \in [1, s) : node[b][i] \geq node[r][k]$$

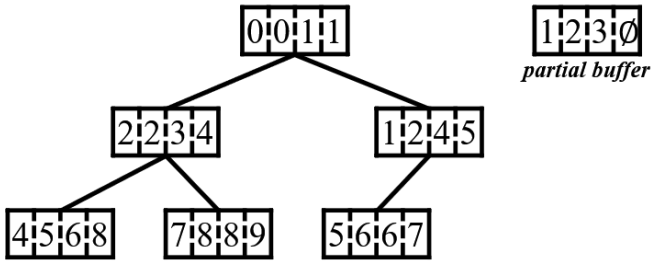
The *generalized heap* is an intuitive extension of the conventional heap. For instance, when $k = 1$, the *generalized heap* becomes a conventional heap where each batch contains exactly one key value. Note that in this case, the *generalized heap properties* still holds: property 1 and 2 are inherently satisfied with the relationship between parent and child node, while properties 3 and 4 do not apply since the size of *partial buffer*, in this case, will be $k - 1 = 0$, which means no *partial buffer* is needed for conventional heap.

### 3.1.2 Array Representation of the Generalized Heap

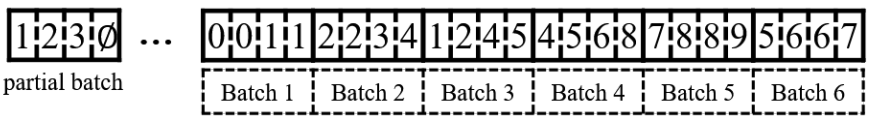
As the most space-efficient approach, an array is used to store and represent the *generalized heap* in memory. Each slot in the array stores one key value and $k$ consecutive slots constitute a data batch. The *generalized heap* then can be represented using many consecutive data batches stored in a linear array with the first $k$ slots contain elements in the root batch, the next $k$ slots contain elements in the next batch, *etc*. Formally, the *generalized heap* can be represented by array using the following rules:

1. The root batch will be stored at Arr[0] through Arr[$k-1$].
2. Suppose the total number of batches is $N$, then for $0 \leq\ i \leq\ N$, indexes of the parent and child batches of the $i^{th}$ batch can be obtained through:
   - Parent($i$) = Arr[$k \times\ ((i-1)/2)$] through Arr[$(k+1) \times\ ((i-1)/2) - 1$]
   - Left($i$) = Arr[$k \times\ ((2 \times\ i)+1)$] through Arr[$(k+1) \times\ ((2 \times\ i)+1) - 1$]
   - Right($i$) = Arr[$k \times\ ((2 \times\ i)+2)$] through Arr[$(k+1) \times\ ((2 \times\ i)+2) - 1$]

Therefore, the array representation store the underling binary tree implicitly using minimal space. Array entries that are in the range of [$i \times\ k$, $(i+1) \times\ k-1$] are stored in $i^{th}$ node in the *generalized heap*. A comparison between tree and array representation of the *generalized heap* with $k$=4 is shown on Fig. 3.1. Note that the *partial buffer* is stored separately in both representations.




(a) Tree representation




(b) Array representation

Figure 3.1: Different representation of the *generalized heap* with $k$ = 4

## 3.2 Operations on Generalized Heap

Resembling the conventional heap, the *generalized heap* allows two basic operations: insertion operation that inserts at most $k$ keys at once, and deletion operation that retrieves k keys from the root node. In both insertion and deletion operations, a **MergeAndSort** process is used to merge and sort key values in two different nodes. Assuming there are two nodes, $l$ and $r$, it takes $2k$ keys from these two nodes as input and returns $2k$ sorted keys where smaller half is stored back to $l$ while the larger half is stored back to in $r$. While the implementation of the MergeAndSort process will be discussed in Chapter 4, the detailed description and examples of the two operations are discussed in the following section.

### 3.2.1 Insert Operation

The insert operation inserts at most $k$ keys at once, for input size larger than $k$, the operation can be divided into multiple concurrent insertion operations. Each insertion starts by inserting a new node at the first available position in the heap. This new node is called the *target* node. Then the path from the root node to the *target* node, the *insert path*, can be found and utilized to propagate the target node to the correct location.

Two types of insertions are allowed in the *generalized heap*: ① *top-down insertion*, which starts the insertion from the root node and propagates in a top-down manner along with the *insert path* until the *target* node is reached; ② *bottom-up* insertion, which starts the insertion from the target node and propagates upward until the *generalized heap properties* are satisfied or the root node is reached.

#### Top-down Insertion

The *top-down* insertion starts the process from the root node and propagates towards the *target* node along the *insert path*. During the propagation, a MergeAndSort operation is performed between the key values in the *target* node and each node it encounters in the *insert path*. The MergeAndSort operation stores the smaller half of key values at the node in the *insert path* while storing the larger half in the *target* node for further propagation. An example of *top-down* insertion is shown in Fig. 3.2 and the pseudo-code is shown in Alg. 5. Note that the

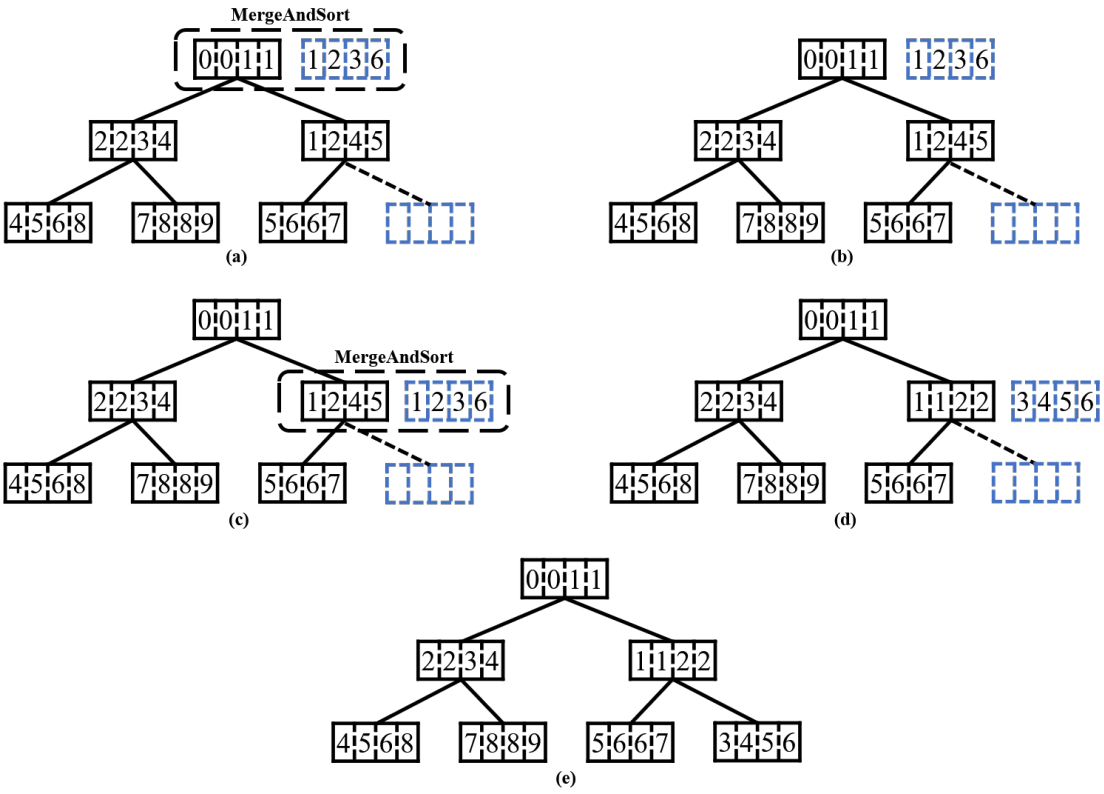



Figure 3.2: Example: *Top-down* Insertion of [1,2,3,6] with $k = 4$

index of the root node stored in the underlying array is 1, thus the propagation starts with *cur* = 1 (line 4).

**Algorithm 5:** *Top-down* Insertion on *the Generalized Heap*

```
1 define B[x] as heap.node[x]
2 Procedure insert_top_down: (ins_items)
3 ins_items = sort(ins_items)
4 tar = heap_size++; cur = 1; level = log2(tar) - 1
5 while cur != tar do
6     (B[cur], ins_items) = MergeAndSort(B[cur], ins_items)
7     cur = tar >> -- level
```

### Bottom-up Insertion

The *bottom-up* insertion starts the process from the *target* node and propagates towards the root node along the *insert path*. In contrast to the *top-down* approach, it is possible to terminate the propagation process in the middle of the heap without traversing every node in the *insert path*

once the *generalized heap properties* are satisfied. During the propagation process of *bottom-up* insertion, a MergeAndSort operation is performed between a node in the *insert path* and its parent node until it reaches the root node or the *generalized heap properties* are satisfied in the middle of the heap. The pseudo-code of the *bottom-up* insertion is shown in Alg. 6. Note the *cur* stores the index of the current node and *par* stores the index of the parent node.

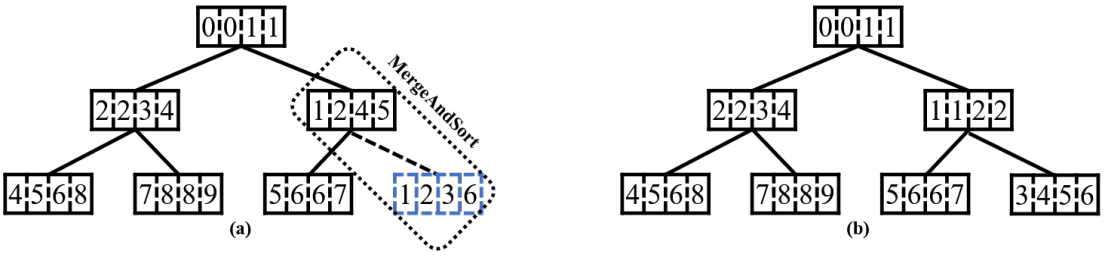



Figure 3.3: Example: *Bottom-up* Insertion of [1,2,3,6] with $k = 4$

**Algorithm 6:** *Bottom-up* Insertion on *the Generalized Heap*

```
1 define B[x] as heap.node[x]
2 Procedure insert_top_down: (ins_items)
3 ins_items = sort(ins_items)
4 cur = heap_size++; par = cur ≫ 1
5 while cur != 1 do
6     if B[cur][0] ≥ B[par][k-1] then
7         break
8     (B[cur], ins_items) = MergeAndSort(B[cur], ins_items)
9     cur = par; par = cur ≫ 1
```

### 3.2.2 Delete Operation

The delete operation retrieves $k$ keys stored in the root node and remove the root node from the heap. A heapify process is needed after deletion since the root node is removed. This process is necessary since the *generalized heap properties* are violated after removing the root node. To make the heap obey the *generalized heap properties* again, the heapify process first moves key values stored in the last leaf node to the root node. Followed by propagating the values in the root node in a *top-down* manner. During the propagation, it will first perform the MergeAndSort operation on two child nodes $l$ and $r$. The output of the MergeAndSort operation consists of $2k$ sorted key values from $l$ and $r$. After the MergerAndSort operation,

the $k$ smallest keys will be placed back to the child node that has a smaller largest key, and the $k$ largest keys will be placed to the other child node. Then, another MergeAndSort operation is performed between the current node and $l$, the $k$ smallest keys from the output will be placed in the current node and the $k$ largest keys will be place in $l$. The propagation process continues until the *generalized heap properties* are satisfied or it reached the leaf node. An example of deletion operation can be found in Fig. 3.4. The pseudo-code of deletion operation can be found in Alg. 7. Note the values in the last node are re-initialized to MAX value after it is moved to the root node to make sure the old keys are covered (line 6).

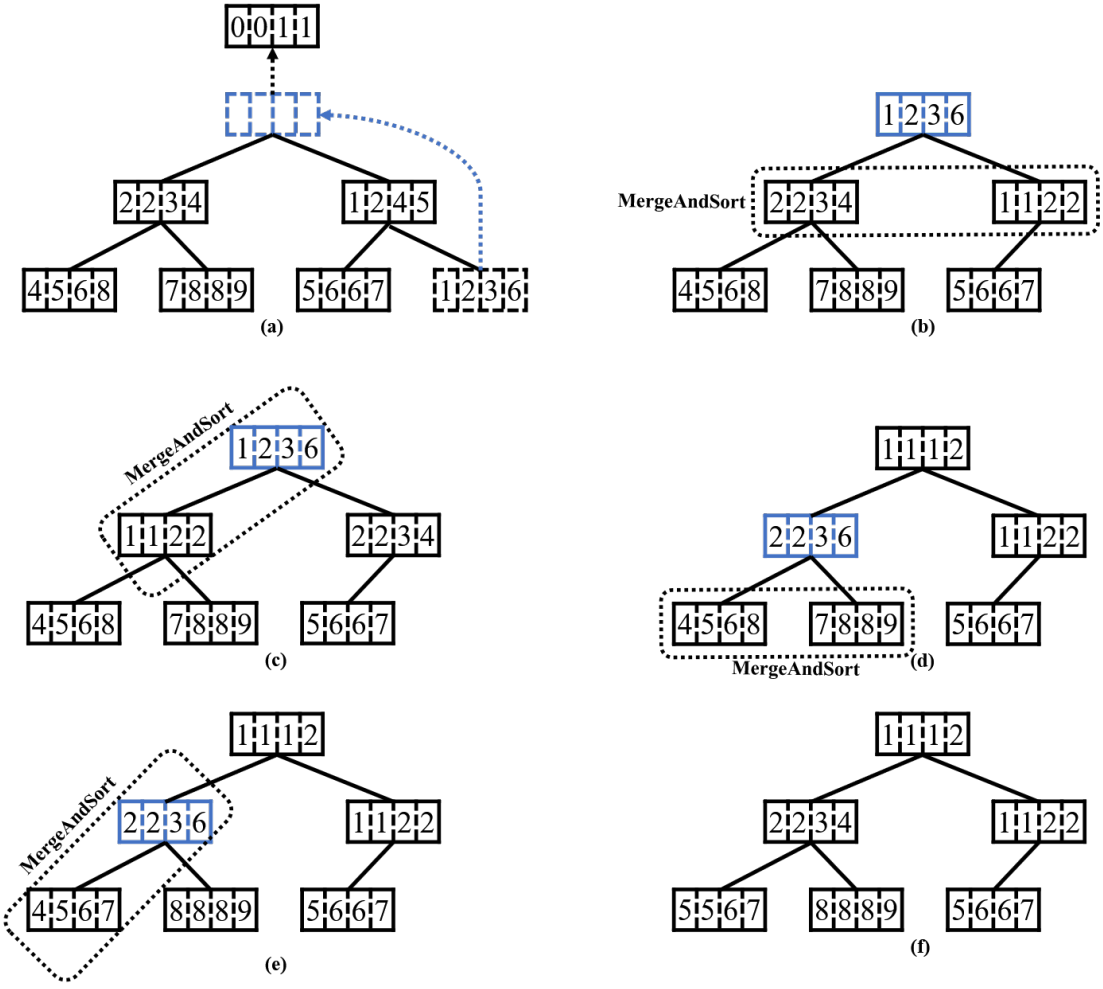



Figure 3.4: Example: Deletion Operation

**Algorithm 7:** Deletion on *the Generalized Heap*

```
1  define B[x] as heap.node[x]
2  Procedure delete: (del_items)
3  if heap.size == 0 then return
4  del_items = B[1]
5  tar = heap.size-- ; cur = 1;
6  B[1] = B[tar]; B[tar] = MAX_VALUE
7  if tar == 1 then return
8  while 1 do
9      l = Left(cur); r = Right(cur)
10     (B[l], B[r]) = MergeAndSort(B[l], B[r])
11     if B[cur][k-1] ≤ B[l][0] == 0 then break
12     (B[cur], B[l]) = MergeAndSort(B[cur], B[l])
13     cur = l
```

### 3.2.3 Discussion

The *bottom-up* insertion approach has a few advantages over the *top-down* approach: ① the number of nodes need to be traversed during the insertion process using the *bottom-up* approach might be less than its *top-down* counterpart since it can terminate earlier. ② When implemented in parallel, locks of nodes are required for each operation. The *bottom-up* implementation reduces lock contentions on higher-level nodes as the workload is distributed to lower-level nodes. However, while implementing the *bottom-up* approach in parallel, one must pay more attention to the potential deadlock caused by the opposite propagation direction of insertion and deletion. The concurrent design of insertion and deletion is discussed in the following section.

## 3.3 Concurrent Generalized Heap

Concurrent insertion and deletion are possible on the *generalized heap*. The parallel insertion and deletion algorithms presented in this thesis are inspired by the methods discussed in [5] and [6], which introduce concurrent insertion and deletion on the conventional heap with *top-down* and *bottom-up* insertion, respectively. In the following sections, we present and discuss the concurrent algorithms of insertion and deletion operations on two types the *generalized heap*:

concurrent *generalized heap* with ① ***T**op-**D**own **Ins**ertion and **T**op-**D**own **D**eletion*, the ***TD-INS/TD-DEL*** Heap and ② ***B**ottom-**U**p **Ins**ertion and **T**op-**D**own **D**eletion*, the ***BU-INS/TD-DEL*** Heap, respectively.

### 3.3.1 Locking Mechanism

To allow the concurrent insertion and deletion operations while ensuring the correctness of the heap, an effective locking mechanism is needed. Instead of locking the entire heap, [5] and [6] applied a simple locking mechanism on the conventional heap where each node in the heap is associated with an individual lock. Thus, multiple propagation flows of heap operations can exist simultaneously, exploiting the inter-node parallelism. We adapt and extend a similar locking mechanism to the concurrent *generalized heap*. In our locking mechanism, the *partial buffer* shares the same lock with the root node while other nodes are protected by their own lock. Fig. 3.5 shows the detailed locking mechanism of concurrent heap operations of the concurrent *generalized heap*. Nodes in the *insert/delete path* are marked $N_1$, $N_2$, ..., $N_k$ where $N_i$ is the parent of $N_{i+1}$. All three operations follow the same *parent-child* locking order, thus avoids having deadlock.

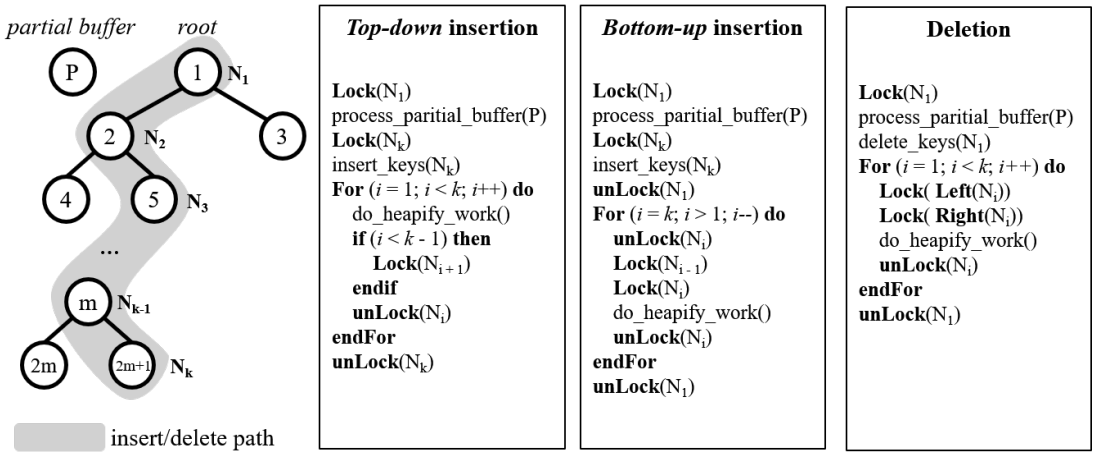



Figure 3.5: Locking Mechanism

The *top-down* insertion starts the propagation at the root node $N_1$ and propagates along with nodes on the *insert path*. It first locks $N_1$ to process the *partial buffer*, then it locks $N_k$ to insert key values to the *target* position and start the *top-down* propagation. During propagation, it acquires the lock of $N_{i+1}$ before releasing the lock of $N_i$, which follows the parent-child locking

order. Though the *bottom-up* insertion has a different propagation direction, it is implemented in a way to follow the same locking pattern as well. Lock of $N_1$ and $N_k$ are acquired to process *partial buffer* and insert key values to *target* position before the propagation starts. During the propagation, it must obtain locks of both $N_i$ and its parent before it starts doing heapify works, so there is no deadlock between threads. The locking order of deletion operation is similar to that of the *top-down* insertion, with the difference being that the delete operation needs to acquire locks for both children of $N_i$ when propagating down. When heapification of $N_i$ is finished, the control of $N_{i+1}$ will remain in the same thread while the lock of $N_i$ and the other child is released. Since all operations follow the same *parent-child* locking order, there will be no conflict in acquiring locks between different threads, thus no deadlock will happen.

**Algorithm 8:** Macros for Locking on the Concurrent Generalized Heap

```
1 define MS_LOCK(x, s_old, s_new) as while CAS(state(x), s_old, s_new)) != s_old
2 define MS_TRYLOCK(x, s_old, s_new) as return CAS(state(x), s_old, s_new)
3 define MS_UNLOCK(x, s_old, s_new) as CAS(state(x), s_old, s_new)
```

Locks in the concurrent *generalized heap* are implemented as *multi-states* locks. Each lock can represent different states of corresponding nodes. Note that the locks of *top-down* insertion and *bottom-up* insertion will have different states due to algorithmic differences. The details of *TD-INS/TD-DEL* Heap and *BU-INS/TD-DEL* Heap will be discussed in the following section. The multi-states locks are implemented using atomicCAS, a well-optimized intrinsic atomic operation on NVIDIA GPUs [14]. The locking related macros presented in the algorithms is shown in Alg. 8.

### 3.3.2 *TD-INS/TD-DEL* Heap

The *TD-INS/TD-DEL* heap supports two operations: *top-down* insertion and *top-down* deletions. The operations are implemented based on the locking mechanism described in Fig. 3.5. The lock states and corresponding meaning of the insert operation on the *TD-INS/TD-DEL* heap are:

- **AVAIL**: This node is available for operations.
- **INUSE**: This node is currently locked by an operation.

- **TARGET**: This node is the *target* node of an insertion operation.
- **MARKED**: This node is needed by a delete operation for cooperating insertion and deletion.

The FSA (Finite-State Automata) of the *TD-INS/TD-DEL* heap is shown in Fig. 3.6. Both insertion and deletion operations are able to lock the control of a heap node by changing its node state from **AVAIL** to **INUSE**.

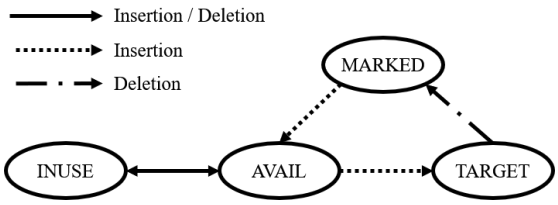



Figure 3.6: FSA of the *TD-INS/TD-DEL* Heap

Note that **MARKED** is a special node state of the *TD-INS/TD-DEL* heap where deletion and insertion can cooperate to speed up the overall propagation[5]. After the deletion process removed the root node, it first checks if the last node is locked by an insertion process as *target* node, if so, it changes the state of the *target* node from **TARGET** to **MARKED**. When an insertion operation finds that the state of the *target* node is changed to **MARKED**, it immediately knows that a deletion operation is trying to move the key values in the last leaf node to the root node. It then terminates the insertion propagation and directly move the keys in the last node to the root node where the deletion operation takes over the control and continues executing. This early termination of the *top-down* insertion operation is valid since the *generalized heap properties* are kept valid except for the *target* node during the insertion propagation. As a result, the overall performance of deletion propagation can be improved by early termination of an in-progress insertion. The pseudo-code for concurrent *top-down* insertion operation is shown in Alg. 9 and the pseudo-code for concurrent deletion operation is shown in Alg. 10.

Note that the *partial buffer* and the root node are deliberately designed to share the same lock. Since any operation that utilized the partial buffer will require access to the root node for comparison, using the same lock guarantees only one thread will work on the partial buffer and the root node at the same time.

For deletion operations, the partial buffer is used only when the heap is empty, *i.e*. the total

number of key values is less than $k$, where all keys are stored in the partial buffer. By the fourth property of the *generalized heap*, the smallest $k$ keys of the generalized heap must be stored in the root node, which means key values stored in the partial buffer will not be the smallest items unless the heap is empty.

On the other hand, to correctly process partial batch insertion, it first applies the MergeAndSort operation between the partial buffer and items to be inserted. If the result of the MergeAndSort operation can not be stored entirely back to the partial buffer, *i.e*. size of partial buffer plus insert size is greater than $k$. Then the $k$ smallest keys from the result of the MergeAndSort operation will be inserted to the heap as a full batch and propagate downwards, while the rest of the keys will be stored back to the partial buffer (Fig. 9 line 5). Otherwise, the result of the MergeAndSort operation will be stored back to the partial buffer and another MergeAndSort operation is applied between the new partial buffer and the root node (line 9), after which the *generalized heap properties* will be satisfied. Though supporting partial batch insertion introduces extra overhead, by maintaining the partial buffer, memory locality and intra-node parallelism can be exploited since the majority of Insert/Delete operations are processing the regular-sized input *i.e*. a full batch with $k$ keys.

### 3.3.3 *BU-INS/TD-DEL* Heap

In the *TD-INS/TD-DEL* heap, all heap operations follow the same *top-down* direction, which results in a considerable contention on the top level of the heap, especially the contention on the root node. In order to alleviate this situation, [6] introduced an insertion algorithm that starts the insertion process from the bottom of the heap while maintaining the same *parent-child* locking order. Their implementation allows the inserting thread to temporarily release the control for items to be inserted and labeled the corresponding inserting thread and items by storing an extra tag. We applied a similar approach to implement the concurrent *bottom-up* insertion for the *generalized heap*. However, our concurrent *bottom-up* insertion implementation does not need to store the extra tag.

Our concurrent *bottom-up* insertion implementation follows the same parent-child locking order described in Fig. 3.5. In contrast to the *top-down* insertion, the *bottom-up* insertion

**Algorithm 9:** *Top-down* Insertion on the *TD-INS/TD-DEL* Heap

```
1  Procedure concurrent_insert_top_down: (ins_items, ins_size)
2  ins_items = sort(ins_items)
3  MS_LOCK(1, AVAIL, INUSE)
4  if pBuffer.size + ins_size ≥ k then
5      (ins_items, pBuffer) = MergeAndSort(ins_items, pBuffer)
6  else
7      (pBuffer, ins_items) = MergeAndSort(ins_items, pBuffer)
8      if heap.size != 0 then
9          (B[1], pBuffer) = MergeAndSort(B[1], pBuffer)
10     MS_UNLOCK(1, INUSE, AVAIL)
11     return
12 target = heap.size++; cur = 1; level = log2(tar) - 1
13 if target != 1 then
14     MS_LOCK(tar, AVAIL, INUSE)
15 while cur != target do
16     if state(tar) == MARKED then break
17     (B[cur], ins_items) = MergeAndSort(B[cur], ins_items)
18     cur = target ≫ -- level
19     if target != cur then
20         MS_LOCK(cur, AVAIL, INUSE)
21     MS_UNLOCK(cur≫1, INUSE, AVAIL)
22 tstate = MS_TRYLOCK(target, TARGET, INUSE)
23 target = (tstate == TARGET ? target: 1)
24 B[target] = ins_items
25 if target != cur then
26     MS_UNLOCK(target, state(target), AVAIL)
27 MS_UNLOCK(cur, INUSE, AVAIL)
```

operation has the following node states:

- **AVAIL**: This node is available for operations.
- **INUSE**: This node is currently locked by an operation.
- **INSHOLD**: This node is temporarily released by an insertion process.
- **DELMOD**: This node is originally in **INSHOLD** state and has been modified by a deletion process.

The FSA of node states of the *BU-INS/TD-DEL* heap is shown in Fig. 3.7. Similar to the both *TD-INS/TD-DEL* operations discussed in the last section, both insertion and deletion operations for the *BU-INS/TD-DEL* heap can change the state of a node from **AVAIL** to **INUSE**

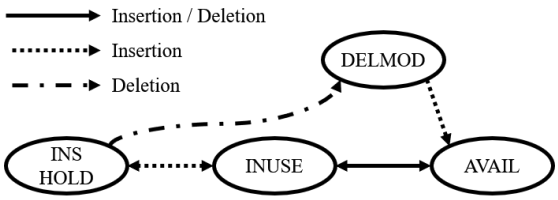



Figure 3.7: FSA of the *BU-INS/TD-DEL* Heap

and the other way around. However, the concurrent *bottom-up* insertion algorithm requires different node states since it will temporarily release the control of the nodes to guarantee the parent-child locking order. After the insertion process finished modifying $N_k$ and try to propagate upwards, it will temporarily unlock $N_k$ and change its state from **INUSE** to **INSHOLD**. Then, the insertion operation will acquire the control of $N_{k-1}$ before re-acquiring $N_k$ again to perform the *bottom-up* propagation. When it successfully re-acquire control of $N_k$, the state of $N_k$ can be in three different cases:

1. **DELMOD**: $N_k$ has been modified by one or more delete operations, the heap properties between $N_k$ and $N_{k-1}$ are satisfied, skip this level and perform MergeAndSort operation between $N_{k-1}$ and $N_{k-2}$.
2. **AVAIL**: $N_k$ is deleted by other processes while it is released, release $N_{k-1}$ and terminate the insertion process.
3. **INSHOLD**: $N_k$ has not been modified by other operations, continue the insertion propagation and the MergeAndSort operation can be performed between $N_k$ and $N_{k-1}$.

The *partial buffer* in the *BU-INS/TD-DEL* heap is handled in the same way as we mentioned in Section 3.3.2. It shares the same lock with the root node and both insertion and deletion operations process the data in the partial buffer while the root node is locked.

**Algorithm 10:** Deletion on the *TD-INS/TD-DEL* Heap

```
1  Procedure concurrent_delete: (delete_items)
2  MS_LOCK(1, AVAIL, INUSE)
3  if heap.size == 0 then
4      if pBuffer.size != 0 then
5          delete_items = pBuffer[1:pBuffer.size]
6      MS_UNLOCK(1, INUSE, AVAIL) return
7  delete_items = B[1]
8  target = heap.size--; cur = 1
9  tstate = MS_TRYLOCK(target, TARGET, MARKED)
10 if tstate == MARKED then
11     while (state(target != AVAIL))
12 else
13     MS_LOCK(target, AVAIL, INUSE)
14     B[1] = B[target]; B[target] = MAX_VALUE
15     MS_UNLOCK(target, INUSE, AVAIL)
16 (B[1], pBuffer) = MergeAndSort(B[1], pBuffer)
17 while 1 do
18     l = Left(cur); r = Right(cur)
19     lstate = INUSE; rstate = INUSE
20     while lstate == INUSE do
21         lstate = MS_TRYLOCK(l, AVAIL, INUSE)
22     if lstate != AVAIL then
23         // cur has no child
24         MS_UNLOCK(cur, INUSE, AVAIL)
25         return
26     while rstate == INUSE do
27         rstate = MS_TRYLOCK(r, AVAIL, INUSE)
28     if rstate != AVAIL then
29         // cur only has left child
30         (B[cur], B[l]) = MergeAndSort(B[cur], B[l])
31         break
32     // Suppose right child has the largest item
33     (B[l], B[r]) = MergeAndSort(B[l], B[r])
34     MS_UNLOCK(r, INUSE, AVAIL)
35     if (B[cur][k - 1] ≤ B[l][0]) then break
36     (B[cur], B[l]) = MergeAndSort(B[cur], B[l])
37     MS_UNLOCK(cur, INUSE, AVAIL)
38     cur = l
39 MS_UNLOCK(cur, INUSE, AVAIL)
40 MS_UNLOCK(l, INUSE, AVAIL)
```

**Algorithm 11:** *Bottom-up* Insertion on the *BU-INS/TD-DEL* Heap

```
1  Procedure concurrent_insert_bottom_up: (insert_items, insert_size)
2  ins_items = sort(ins_itmes)
3  MS_LOCK(1, AVAIL, INUSE)
4  if pBuffer.size + ins_size ≥ k then
5      (ins_items, pBuffer) = MergeAndSort(ins_items, pBuffer)
6  else
7      (pBuffer, ins_items) = MergeAndSort(ins_items, pBuffer)
8      if heap.size != 0 then
9          (B[1], pBuffer) = MergeAndSort(B[1], pBuffer)
10     MS_UNLOCK(1, INUSE, AVAIL)
11     return
12 cur = heap.size++; parent = cur ≫ 1
13 if cur != 1 then
14     MS_LOCK(cur, AVAIL, INUSE)
15     MS_UNLOCK(1, INUSE, AVAIL)
16 B[cur] = ins_items
17 while cur != 1 do
18     MS_UNLOCK(cur, INUSE, INSHOLD)
19     pstate = INUSE; cstate = INUSE
20     while pstate == INUSE || pstate == INSHOLD do
21         pstate = MS_TRYLOCK(parent, AVAIL, INUSE)
22     if (pstate != AVAIL) then return
23     while cstate == INUSE do
24         cstate = MS_TRYLOCK(cur, INSHOLD, INUSE)
25     if cstate == DELMOD then
26         MS_UNLOCK(cur, DELMOD, AVAIL)
27     else if cstate == AVAIL then
28         MS_UNLOCK(parent, DELMOD, AVAIL)
29         return
30     else
31         if B[cur][0] ≥ B[parent][k - 1] then
32             MS_UNLOCK(parent, INUSE, AVAIL)
33             break
34         B[parent], B[cur] = MergeAndSort(B[parent], B[cur])
35         MS_UNLOCK(cur, INUSE, AVAIL)
36     cur = parent; parent = cur ≫ 1
37 MS_UNLOCK(cur, INUSE, AVAIL)
```

**Algorithm 12:** Deletion the *BU-INS/TD-DEL* Heap

```
1  Procedure concurrent_delete: (delete_items)
2  MS_LOCK(1, AVAIL, INUSE)
3  if heap.size == 0 then
4      if pBuffer.size != 0 then
5          delete_items = pBuffer[1:pBuffer.size]
6          pBuffer = MAX_VALUES
7      MS_UNLOCK(1, INUSE, AVAIL)
8      return
9  delete_items = B[1]
10 target = heap.size--
11 MS_LOCK(target, AVAIL, INUSE)
12 B[1] = B[target]; B[target] = MAX_VALUES
13 MS_UNLOCK(target, INUSE, AVAIL)
14 (B[1], pBuffer) = MergeAndSort(B[1], pBuffer)
15 cur = 1; cstate = INUSE
16 while True do
17     l = Left(cur); r = Right(cur)
18     lstate = INUSE; rstate = INUSE
19     while lstate == INUSE do
20         lstate = MS_TRYLOCK(l, state(l, INUSE)
21     while rstate == INUSE do
22         rstate = MS_TRYLOCK(r, state(r, INUSE)
23     lstate = (lstate == INSHOLD ? DELMOD : lstate)
24     rstate = (rstate == INSHOLD ? DELMOD : rstate)
25     // Suppose the right child batch has the largest item
26     (B[l], B[r]) = MergeAndSort(B[l], B[r])
27     MS_UNLOCK(r, INUSE, rstate)
28     if B[cur][k - 1] ≤ B[l][0] then
29         MS_UNLOCK(cur, INUSE, cstate)
30         MS_UNLOCK(l, INUSE, lstate)
31         return
32     B[cur], B[l] = MergeAndSort(B[cur], B[l])
33     MS_UNLOCK(cur, INUSE, cstate)
34     cur = l; cstate = lstate
```

# Chapter 4

# Implementation

In our implementation, the concurrent *generalized heap* is based on thread-block-level parallelism since block-level barrier synchronization is natively supported in NVIDIA GPUs while no built-in thread-level synchronization is provided and synchronization between thread blocks introduce extra overhead. Threads in one thread block can cooperate for one insertion or deletion operation. Since threads within the same block have accesses to the same shared memory space, block-level operations can exploit the performance of shared memory through data reusing during propagation. Furthermore, since key values of heap node are stored consecutively in the memory, block-level operations also take advantage of memory coalescing. The *multi-state* lock introduced in Section 3.3.1 is safer to implement as a block-level lock since explicit block-level synchronization is available while desynchronization within thread warp might result in deadlock[15].

Heap operation of the *generalized heap* is implemented through the following basic building blocks: ① parallel sorting operation; ② the MergeAndSort operation. We will discuss these building blocks respectively in the following sections along with some optimization techniques that apply to the implementation of the concurrent *generalized heap*.

## 4.1 Sorting Operation

Both *top-down* and the *bottom-up* the insertion operation will sort the input values before initiating the propagation process. We refer to this sorting operation as the *local* sorting since it is independent of the heap nodes. To perform the local sorting, input values are first loaded to the shared memory for efficient manipulation and movement. Since the capacity of shared memory per thread block is limited by the hardware, the input size of each insertion operation is also

limited (no more than 1K pairs in our test case). By setting the node capacity, $k$ (recall that input with more than $k$ values will be divided into multiple insertions with at most $k$ values), the shared memory can be fully utilized by setting the right $k$.

The local sorting on the shared memory is implemented using parallel bitonic sorting. Bitonic sorting is a comparison-based sorting algorithm optimized for parallel computing. While the analysis of bitonic sorting is beyond the scope of this thesis, it can be found in [16]. In comparison with other efficient sorting algorithms on GPUs such as the parallel radix sort, the complexity of bitonic sort only related to the number of input items instead of the size(length) of data. Moreover, other sorting algorithms like the radix sort require the data to have the same lexicographical order as integers, which makes it less generic for different cases. Since the input size of the insert operation is limited by $k$, which is upper-bounded a relatively small number, and the fact that it supports any generic data types, parallel bitonic sorting algorithm whose complexity depends on the number of input items is more suitable for the local sorting in the *generalized heap*.

## 4.2 MergeAndSort Operation

The MergeAndSort operation is extensively used in both insertion and deletion operations to heapify the sub-heap. The MergeAndSort operation merges $2k$ items from two lists of size $k$, $l$ and $r$, and store the $k$ smallest items back to $l$ and $k$ largest items back to $r$. List with a size smaller than $k$ will be appended with MAX_VALUE to extend the list to size $k$. Since key values stored within a node are already sorted by the local sorting operation, the MergeAndSort operation does not need to explicitly sort the lists again. Instead, the GPU *merge-path* algorithm[17] can be used to merge two sorted lists. The merge-path algorithm takes advantage of GPU architecture by distributing workload evenly to threads, minimized the overhead of load-imbalance. Moreover, it was implemented with low-latency and high-bandwidth shared memory usage which is suitable for our concurrent insertion/deletion operations as data will be loaded to shared memory in both operations. Detailed description and analysis for the GPU merge-path algorithm can be found in [17].

## 4.3 Optimization

Various optimization techniques are applied to the concurrent operations of the *generalized heap:*

### 4.3.1 Remove Redundant MergeAndSort Operations

The heapify process through repeatedly applying MergeAndSort operations contributed heavily to the overhead of the algorithm. Through propagating with the MergeAndSort operation, the *generalized heap properties* are guaranteed to be satisfied at each level of propagation. Fortunately, it is possible to reduce the number of MergeAndSort operations performed. In our implementation, a comparison is made between keys in the nodes before a MergeAndSort operation is performed. If the largest key in a node is smaller than the smallest key of the other node, instead of performing a MergeAndSort operation, a simple swap between two nodes is sufficient since items in the nodes are already sorted. As a result, the number of MergeAndSort operation performed is reduced within every insertion and deletion operations.

### 4.3.2 Early Termination

Early termination can be applied to operations of both *TD-INS/TD-DEL* heap and *BU-INS/TD-DEL* heap. For the *BU-INS/TD-DEL* heap, both the deletion and the *bottom-up* insertion operation can terminate before it traverses through all nodes in the *Insert/Delete path* when the *generalized heap properties* are satisfied. On the other hand, only deletion operation for the *TD-INS/TD-DEL* heap can terminate early since *top-down* insertion requires the algorithm to propagate the key values through the whole *insert path* to the *target* node. This optimization reduces the number of heap level each operation will propagate through. Thus increases the throughput of the model.

### 4.3.3 Bit-Reversal Permutation

Each insertion operation starts with deciding a *target* node. If consecutive insertions choose a neighboring *target* node, their *insert path* will be highly overlapped, which might result in serious contention during propagation. To avoid this situation, a bit-reversal permutation[6]

technique is applied to make sure that the *insert path* of two consecutive insertion operation will share no common nodes except the root node. This technique can also be applied to deletion operation where a target node is selected to be moved to the root position after the root is removed. The *target* node of deletion operation is selected following the bit-reversal permutation in insertion operation but in the reverse order.

# Chapter 5

# Evaluation and Analysis

A comprehensive evaluation is performed on the concurrent *generalized heap*. Experiments regarding the performance of the model are conducted focusing on six perspectives:

- How does the concurrent generalized heap performs compared to sequential CPU heap and a previous implementation of GPU heap [7] under different access patterns.
- How does the different number of active insertion or deletion operations (in terms of the number of active thread blocks) affect contention levels and scalability of the model.
- How does the node capacity $k$ affect the performance of each operation with respect to different block size.
- How does the percentage of partial batch insertion affects the overall performance.
- How would the performance of concurrent operations varies under different heap utilization.
- How does the model perform on real-world applications: SSSP(Single-Source-Shortest-Path) and 0/1 knapsack problems.

## 5.1 Experiment Setup

The experiments are conducted on an Nvidia TITAN X GPU with an Intel Xeon E5-2620 CPU. The Xeon E5-2620 CPU is working on 2.1 GHz frequency. The TITAN X GPU has 28 streaming multi-processors (SMs) with 128 cores, for a total of 3584 cores. Every thread block can allocate at most 48 KB of shared memory and 64K available registers. The maximum number of active threads per block is 1K, the maximum number of active threads per SM is 2K.

Table 5.1: Comparison of Heap Performance on CPU v.s. GPU

| Method | Input Type | | |
|---|---|---|---|
| | **Randomized(ms)** | **Ascending(ms)** | **Descending(ms)** |
| **STL Heap** | 1,959,550 | 1,214,906 | 1,898,015 |
| **P-sync Heap** | 209,648 | 201,66 | 205,761 |
| **TD-INS/TD-DEL Heap** | 112,090 | 99,082 | 100,163 |
| **BU-INS/TD-DEL Heap** | 104,417 | 96,247 | 97,593 |

Test Configurations: Number of Thread Blocks: 128,
Size of Thread Block: 512, $k$ = 1024, Number of Keys: 512M

## 5.2 Concurrent Heap v.s. CPU Heap and GPU Baseline

The performance of the concurrent *generalized heap* is compared with a CPU heap implementation and a GPU baseline implementation. We use C++ STL priority queue library as the baseline CPU heap, which is referred to as the *STL Heap*. We use previous GPU heap implementation by He, Deo, and Prasad [7] as the GPU baseline, which is referred to as the *parallel synchronous heap* or in short, the *P-Sync Heap*. Note that the insertion operation of the *P-Sync Heap* is implemented in s *top-down* manner while the *STL Heap* is implemented in a *bottom-up* approach. A synthetic test of inserting 512M keys to an empty heap then deleting all 512M keys from the heap is performed. The following input key types are generated and tested: ① randomized 32-bit integers; ② randomized 32-bit integers sorted in ascending order; ③ randomized 32-bit integers sorted in descending order.

The test result is shown in Table 5.1. The concurrent *generalized heap* achieved an average 16.59x speedup compared to the *STL Heap* and 2.03x speedup compared to the *P-Sync Heap*. The best performance of all heap models can be observed when the input keys are sorted in ascending order. This is because for models implemented with *bottom-up* insertion, the *STL Heap* and *BU-INS/TD-DEL Heap*, the insertion operation only needs to place the new item to the *target* node, no propagation is needed since the input is already sorted. While for models implemented with *top-down* insertion, the *P-Sync Heap* and the *TD-INS/TD-DEL Heap*, since the input is sorted, the number of merging operations can be reduced as it can be replaced with simple swap operations between nodes. Both *TD-INS/TD-DEL Heap* and *BU-INS/TD-DEL*

*Heap* perform better than the *P-Sync Heap*. One of the main difference between *P-Sync Heap* and the concurrent *generalized heap* is that the *P-Sync Heap* only allows one insertion or deletion operation to work on the same heap level whereas our implementation of the concurrent *generalized heap* supports concurrent insertion and deletion operations to work simultaneously, which exploits inter-node parallelism. In the following experiments except for integration with real-world applications, randomized 32-bit integers will be used as input to the test.

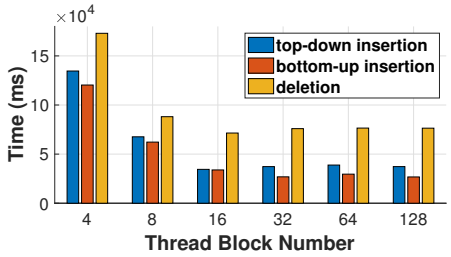



Figure 5.1: Heap Performance w.r.t Number of Blocks

## 5.3 Sensitivity to Number of Thread Blocks

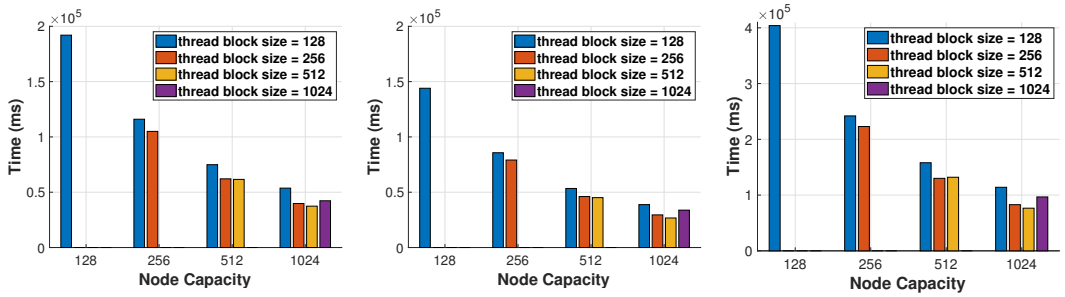
The performance of the concurrent *generalized heap* is evaluated against a different number of thread blocks. Since threads in one thread block can cooperate on one operation at a time, the number of thread block determined the maximum number of concurrent operations possible. However, when the concurrency is increased, the contention level of the heap is also increased. In this experiment, all parameter except the number of blocks is fixed: We set the block size to 512, $k$ = 1024. For *top-down* insertion in *TD-INS/TD-DEL Heap* and *bottom-up* insertion in *BU-INS/TD-DEL Heap* insertion, 512M keys is inserted into the heap. Since both variants of the *generalized heap* implement *top-down* deletion, only deletion operation from the *BU-INS/TD-DEL Heap* is tested as all keys are deleted from a heap containing 512M keys. The result is shown in Figure 5.1.

All heap operations tested exhibit performance increase as the number of blocks increases since more concurrent operations are made possible. However, the increase of performance will be capped as the number of blocks keeps increasing as more contention on the heap nodes are introduced. Note that the deletion operation is always slower than insertions because at every level of the delete path, the process needs to lock three nodes (the parent node and two

child node) and perform at most two MergeAndSort operations compared to two nodes and at most one MergeAndSort operation for the insertion operations. The deletion operation is on average 2.6x slower than insertion operations. Note that the *bottom-up* insertion operation is always faster than *top-down* insertion since the *bottom-up* insertion incurs less contention on higher-level of the heap and it might terminate the propagation process earlier than *top-down* insertion as the described in Section 4.3.2.

## 5.4 Sensitivity to Node Capacity

The performance of the concurrent *generalized heap* is evaluated against a different number of node capacity $k$ with a different number of thread blocks. Due to hardware limitation to the size of shared memory per block, the maximum $k$ is set to be 1K. Furthermore, since it does not make sense to have more than one thread processing one key in the node, the number of threads per block needs to greater than the node capacity. In this experiment, we test the performance of the concurrent *generalized heap* with different node capacity and different thread block size via inserting 512M keys into an empty heap then deleting all keys from the same heap. The test result is shown in Figure 5.2.




(a) *top-down* insertions (b) *bottom-up* insertions (c) deletion operations

Figure 5.2: Heap performance w.r.t Node Capacity and Thread Block Size

As illustrated in Figure 5.2, when the size of the thread block is fixed, the performance of the heap increases as the node capacity $k$ increases. This is because ① for the same number of keys, a larger node capacity means that the heap has fewer levels since each node could store more keys, thus the length of *insert/delete path* would be shorter, result in faster heap operations; ② increasing node capacity also provides more intra-node parallelism on local

operations like the sorting operation. However, Figure 5.2 also shows that it is not always good to increase the thread block size as it could increase the overhead of synchronization within a thread block. We will use thread block size = 512 and $k$ = 1024 for later experiments as they yield the best performance for both insertion and deletion operations.

## 5.5 Sensitivity to Percentage of Partial Batch Insertion

The heap performance is evaluated with different percentages of partial batch insertions. In this experiment, 512M keys are inserted into an empty heap. A portion of 512M keys is inserted as full batches (repeatedly inserting nodes of size $k$), while the rest of the keys are inserted as partial batches with randomly generated sizes. The test result is shown in Figure 5.3. The percentage of full batch insertions can have up to 4x impact on the heap performance. Compared to full batch insertions, inserting with partial batch will need to invoke more insertion operations to insert the same amount of keys, which intensify the contention on the root node since in both *top-down* and *bottom-up* implementation requires the root node to be locked by each insertion process before the corresponding partial batch can be inserted. This phenomenon implies that even though partial batch insertion is allowed in the concurrent *generalized heap*, it would be a nice practice pre-process input with a buffer and avoid frequent partial batch insertion when applying the concurrent *generalized heap* to real-world applications.

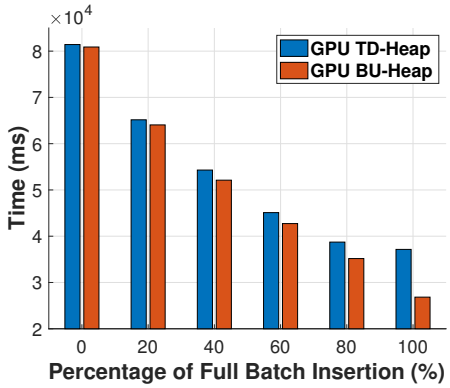



Figure 5.3: Heap Performance w.r.t Percentage of Full Batch Insertion

## 5.6 Sensitivity to Initial Heap Utilization

The performance of heap operations is related to the level of the heap, which determines the length of the *insert/delete path*. In this experiment, we evaluated the difference of performance for the concurrent *generalized heap* with different initial utilization. Dummy keys corresponding to each heap levels are pre-inserted into the heap. For example, $k \times 2^6$ dummy keys are pre-inserted to the heap for an experiment with 6-level initial heap utilization. We use 128 thread blocks to insert 256M items then delete 256M items from a given heap with different utilization. We show the heap performance with respect to different initial heap levels, ranging from 6 to 18, which corresponds to 64K to 256M initial items, in Figure 5.4. It demonstrated that the initial level of the heap will affect the performance of insertion and deletion of the same number of items as both insertion and deletion operations need to traverses more nodes along the *insert/delete path*, and more MergeAndSort operations are performed before the *generalized heap properties* get satisfied. Again, the *BU-INS/TD-DEL Heap* performs better than *TD-INS/TD-DEL Heap* under the same workload.

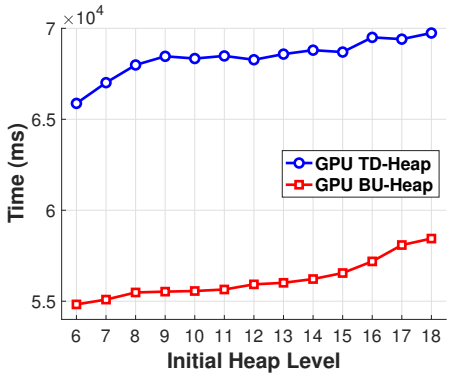



Figure 5.4: Performance w.r.t Initial Heap Size

## 5.7 Applications of Concurrent Heap

The performance of the concurrent *generalized heap* is evaluated under two real-world applications: the SSSP (Single Source Shortest Path) and the 0/1 Knapsack Problem. Both applications rely on an efficient implementation of the priority queue to solve the underlying problem. By processing items higher priority first, the search space of the problem can be reduced thus result

in better performance. Though we only apply the concurrent *generalized heap* to two applications, there are many real-world applications can take advantage of this technique to speed up computation. The purpose of is section is to show the potential of applying the concurrent *generalized heap* to speed up real-world applications on many-core architectures. Further optimization and integration of application-specific heap operations such as asynchronous updates are possible, however, we will leave it as future work.

### 5.7.1 SSSP (Single Source Shortest Path) Problem

As described in Section 2.2.2, the compute-advance model to solve SSSP requires that in every iteration of the SSSP algorithm, nodes are classified as active nodes and inactive nodes based on a comparison between the new distance and the old distance of the node. If the new distance is smaller than the old distance, the corresponding node will be classified as an active node, otherwise, it will be classified as an inactive node. After that, only active nodes will be processed in the next iteration since the distance of inactive nodes will not affect the final result.

We use Gunrock [13], a reputable parallel iterative graph processing library on GPU, as the baseline of this experiment. Gunrock applied the compute-advance model described above to solve for SSSP. We implemented the SSSP solver in the same way with the difference being that after each iteration, active nodes are stored in the concurrent *generalized heap* with their current distance as key values, then the nodes are deleted from the heap in the next iteration for processing. Doing such imposes a relationship between the processing order of each active node where active nodes with shorter distance will be processed before others. Thus the number of active nodes being explored and the overhead spend on unnecessary updates can be reduced.

Since incorporating the heap incurs overhead for heap operations, it is only beneficial to use the heap when a certain threshold is met. In this experiment, we set the threshold $N$ = 10K such that only when the number of active nodes in the current iteration is greater than $N$, will we apply the concurrent *generalized heap* to the SSSP algorithm. 14 real-world graphs are chosen for this experiment and the corresponding graph properties are listed in Table 5.2. The test result is shown in Table 5.3. On average, the heap-based SSSP solver achieve 1.13X speedup for all the graph we tested compared to the baseline. Note that *Stanford_Berkeley* is a

relatively small graph where the number of active nodes in each iteration is not large enough for the improvement in performance to cover the overhead of heap operations.

Timing for different components of the SSSP algorithm is also shown in Table 5.3. The compute time includes the time spend on SSSP computations such as node expanding, distance updating and edge filtering. The heap time includes the time spend on heap operations. The number of nodes visited showed the total number of nodes expanded during the SSSP computation. It is clear that for large-scale applications, applying the concurrent *generalized heap* to the SSSP algorithm reduces the number of nodes expanded significantly, which leads to a reduction in compute time. Even though the incorporation of the concurrent *generalized heap* incurs overhead, the improvement provided by the heap can easily cover the heap overhead with speedups.

Table 5.2: Graph Information

| Graph Name | # Nodes | # Edges | Type of Graph |
|---|---|---|---|
| **AS365** | 3,799,275 | 22,736,152 | 2D FE triangular meshes |
| **bundle_adj** | 513,351 | 20,721,402 | Bundle adjustment problem |
| **coPapersDBLP** | 540,486 | 30,491,458 | DIMACS10 set |
| **delaunay_n22** | 4,194,304 | 25,165,738 | DIMACS10 set |
| **hollywood-2009** | 1,139,905 | 115,031,232 | Graph of movie actors |
| **Hook_1498** | 1,498,023 | 62,415,468 | 3D mechanical problem |
| **kron_g500_logn20** | 1,048,576 | 89,240,544 | DIMACS10 set |
| **Stanford_Berkeley** | 685,230 | 7,600,595 | Berkeley-Stanford web graph |
| **Long_Coup_dt0** | 1,470,152 | 88,559,144 | Coupled consolidation problem |
| **M6** | 3,501,776 | 21,003,872 | 2D FE triangular meshes |
| **NLR** | 4,163,763 | 24,975,952 | 2D FE triangular meshes |
| **rgg_n_2_20_s0** | 1,048,576 | 13,783,240 | Undirected Random Graph |
| **Serena** | 1,391,349 | 65,923,050 | Structural Problem |

### 5.7.2 0/1 Knapsack Problem

The knapsack problem also appears in many real-world applications such as the selection of investments. The knapsack problem is defined as: given a knapsack, weights, and profits for some items with a weight capacity $W$, find the maximum profit we can obtain through putting different combinations of items in the knapsack. The 0/1 knapsack problem is a category of knapsack problem where one must either pick the full item or discard it, no partial selection is allowed. As described in Section 2.2.2, a simple sequential implementation of the knapsack

Table 5.3: Parallel Single Source Shortest Path Performance

| Graphs | Baseline | | Heap Based SSSP w/ N=10K | | | | Speedup |
|---|---|---|---|---|---|---|---|
| | Computate Time(ms) | # Nodes Visited | Heap Time(ms) | Compute Time(ms) | Total Time(ms) | # Nodes Visited | |
| **AS365** | 654.44 | 19,664,769 | 193.43 | 422.12 | 615.55 | 11,843,368 | 1.06 |
| **bundle_adj** | 144.54 | 903,097 | 11 | 126.48 | 137.48 | 877,675 | 1.05 |
| **coPapersDBLP** | 46.13 | 981,876 | 12.52 | 25.46 | 37.98 | 710,794 | 1.21 |
| **delaunay_n22** | 1125.93 | 29,832,633 | 283.04 | 647.61 | 930.65 | 18,607,590 | 1.21 |
| **hollywood-2009** | 100.17 | 2,007,447 | 14.22 | 74.35 | 88.58 | 1,370,459 | 1.13 |
| **Hook_1498** | 233.76 | 2,786,271 | 31.39 | 182.52 | 213.91 | 1,756,776 | 1.09 |
| **kron_g500_logn20** | 117.79 | 2,590,570 | 28.72 | 73.1 | 101.82 | 860,552 | 1.16 |
| **Long_Coup_dt0** | 190.06 | 2,699,927 | 43.46 | 116.77 | 160.23 | 1,571,565 | 1.19 |
| **Stanford_Berkeley** | 55.3 | 530,294 | 5.17 | 52.54 | 57.71 | 462,860 | 0.96 |
| **M6** | 677.95 | 20,972,903 | 161.88 | 472.67 | 634.56 | 16,126,697 | 1.07 |
| **NLR** | 894.71 | 29,583,224 | 318.35 | 439.61 | 757.96 | 16,123,803 | 1.18 |
| **rgg_n_2_20_s0** | 920.27 | 7,112,685 | 46.44 | 701.78 | 748.22 | 5,871,411 | 1.23 |
| **Serena** | 124.16 | 2,594,858 | 27.98 | 84.94 | 112.93 | 1,498,836 | 1.1 |

problem is to enqueue to-explore items into a priority queue with a calculated profit bound as key and explore those with higher priority first. If the global maximum is updated by a high priority item, then we can ignore those items with a profit bound that is lower than the global maximum, thus reducing the search space and speeds up the algorithm. We implemented a parallel algorithm based on the sequential branch and bound algorithm to solve the knapsack problem. Since parallel exploration might result in unnecessary growth to the heap size, we also implemented a technique that filters and discards invalid items in the heap when the heap size is greater than a threshold. We refer to this optimized knapsack solver as knapsack with garbage collection (GC).

S. Martello *et al*. [18] defined and evaluated different categories of the knapsack problems. Using the same generator, we generate 12 knapsack instances that correspond to three *strongly correlated*, three *almost strongly correlated*, three *even-odd strongly correlated*, and three *subset-sum* instances. The properties of these instances are presented in Table 5.4.

In this experiment, we compared the performance between the sequential (CPU) branch and bound algorithm and GPU knapsack algorithm with the concurrent *generalized heap*. The test result is shown in Table 5.5. On average, there is a 2.31X speedup for GPU knapsack and 2.48X for GPU knapsack with GC. Note that GPU knapsack with heap performs especially well on *sumset sum* (ss) instances with a maximum speedup of 12.19X. Since the branch and bound

algorithm is a greedy algorithm, it is possible for it to encounter several local maxima before reaching the global one. Performing parallel exploration alleviates this issue by simultaneously exploring multiple solutions to the problem, which can lead to faster convergence to the global optimum solution.

Table 5.4: Datasets for 0/1 Knapsack Problem

| Dataset | Type | Size | Range |
|---|---|---|---|
| **ks_sc_700_18k** | Strongly Correlated | 700 | 18000 |
| **ks_sc_800_18k** | Strongly Correlated | 800 | 18000 |
| **ks_sc_200_7k** | Strongly Correlated | 200 | 7000 |
| **ks_asc_750_16k** | Almost Strongly Correlated | 750 | 16000 |
| **ks_asc_1300_6k** | Almost Strongly Correlated | 1300 | 6000 |
| **ks_asc_500_7k** | Almost Strongly Correlated | 500 | 7000 |
| **ks_esc_900_18k** | Even-odd Strongly Correlated | 900 | 18000 |
| **ks_esc_1200_13k** | Even-odd Strongly Correlated | 1200 | 13000 |
| **ks_esc_400_8k** | Even-odd Strongly Correlated | 400 | 8000 |
| **ks_ss_100_18k** | Subset Sum | 100 | 18000 |
| **ks_ss_1250_12k** | Subset Sum | 1250 | 12000 |
| **ks_ss_1300_14k** | Subset Sum | 1300 | 14000 |

Table 5.5: 0/1 Knapsack Problem with the *generalized heap*

| | **CPU w/ Priority Queue** | | **GPU w/ concurrent heap** | | | **GPU w/ concurrent heap and GC.** | | |
|---|---|---|---|---|---|---|---|---|
| **Dataset** | **Time (ms)** | **# Nodes Explored** | **Time (ms)** | **# Nodes Explored** | **SpeedUp** | **Time (ms)** | **# Nodes Explored** | **SpeedUp** |
| **ks_sc_700_18k** | 825.70 | 782802 | 670.57 | 813089 | 1.23 | 595.58 | 810717 | 1.39 |
| **ks_sc_800_18k** | 977.49 | 923255 | 757.06 | 955374 | 1.29 | 708.02 | 956514 | 1.38 |
| **ks_sc_200_7k** | 202.40 | 243106 | 199.76 | 249373 | 1.01 | 205.27 | 249935 | 0.99 |
| **ks_asc_750_16k** | 757.17 | 709267 | 722.90 | 389249 | 1.05 | 566.17 | 445231 | 1.34 |
| **ks_asc_1300_6k** | 5239.97 | 4934552 | 5118.03 | 2832737 | 1.02 | 4115.55 | 2404241 | 1.27 |
| **ks_asc_500_7k** | 502.37 | 475402 | 549.10 | 296824 | 0.91 | 499.03 | 295951 | 1.01 |
| **ks_esc_900_18k** | 1128.4 | 1080182 | 848.65 | 1123920 | 1.33 | 796.58 | 1124880 | 1.42 |
| **ks_esc_1200_13k** | 2013.06 | 1950260 | 2066.64 | 2002002 | 0.97 | 1357.75 | 1770747 | 1.48 |
| **ks_esc_400_8k** | 355.25 | 399185 | 346.53 | 418925 | 1.03 | 348.52 | 421504 | 1.02 |
| **ks_ss_100_18k** | 42.38 | 54278 | 3.55 | 55 | 3.48 | 11.92 | 55 | 12.19 |
| **ks_ss_1250_12k** | 20.27 | 23886 | 4.30 | 94 | 4.12 | 4.72 | 94 | 4.92 |
| **ks_ss_1300_14k** | 25.02 | 25305 | 19.64 | 9452 | 1.27 | 18.01 | 8528 | 1.39 |

# Chapter 6

# Conclusion

Parallel computation on many-core architectures benefit various applications. However, arguably the only way to unlock the full potential of many-core architecture for applications that require complex asynchronous communication patterns is through efficient implementation of concurrent data structures. In this thesis, we present a model, the concurrent *generalized heap*, for building concurrent priority queue on many-core architectures and perform thorough evaluations. The concurrent *generalized heap* exploits both intra-node and inter-node parallelism of its underlying tree-based structure. It supports concurrent execution of insertion and deletion operations. Furthermore, any comparable generic data types can take advantage of the concurrent *generalized heap*. Experiments show that the concurrent *generalized heap* is capable of 19.49X speedup compare to the sequential heap on CPU, and 2.11X speedup compared to existing GPU implementation [7]. We also apply the concurrent *generalized heap* to two real-world applications: SSSP and knapsack, which shows an average speedup of 1.13X and 2.48X, respectively.

This thesis sheds light on the potential of incorporating concurrent *generalized heap* with many-core architectures to solve real-world applications. For example, Prim's spanning tree algorithm, data compression and A* search in artificial intelligence which are building blocks for complex system software such as operating system scheduler, interruption handler, and garbage collection. Moreover, since the concurrent *generalized heap* model shares the same operations (insertion and deletion) with the conventional heap, it can be easily integrated into existing frameworks. The future work remains to be integrating the model into more real-world applications. The model presented in this thesis is just a prototype. Application-based optimization can be done to adapt the model case by case such as the garbage collection technique we presented when applying the model to solve the knapsack problem.

# References


[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, Dec 1959.

[2] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, pp. 1389–1401, Nov 1957.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.

[4] I. Buck, "Gpu computing: Programming a massively parallel processor," in *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 17–17, March 2007.

[5] R. Nageshwara and V. Kumar, "Concurrent access of priority queues," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1657–1665, 1988.

[6] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott, "An efficient algorithm for concurrent priority queue heaps," *Information Processing Letters*, vol. 60, no. 3, pp. 151–157, 1996.

[7] X. He, D. Agarwal, and S. K. Prasad, "Design and implementation of a parallel priority queue on many-core architectures," in *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–10, IEEE, 2012.

[8] N. Deo and S. Prasad, "Parallel heap: An optimal parallel priority queue," *The Journal of Supercomputing*, vol. 6, no. 1, pp. 87–98, 1992.

[9] N. Corporation, "Cuda c programming guide, version 9.1," *NVIDIA Corporation, Santa Clara, CA*, 2017.

[10] N. Corporation, "Nvidia fermi compute architecture whitepaper," 2009.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[12] W. Commons, "File:software-perspective for thread block.jpg — wikimedia commons, the free media repository," 2016. [Online; accessed 23-April-2019].

[13] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *ACM SIGPLAN Notices*, vol. 51, p. 11, ACM, 2016.

[14] N. Corporation, "Nvidia's next generation cuda compute architecture: kepler gk110: The fastest, most efficient hpc architecture ever built."

[15] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pp. 235–246, March 2010.

[16] H. W. Lang and F. U. of Applied Sciences, "Sorting networks," Jun 2018.

[17] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 331–340, ACM, 2012.

[18] S. Martello, D. Pisinger, and P. Toth, "Dynamic programming and strong bounds for the 0-1 knapsack problem," *Management Science*, vol. 45, no. 3, pp. 414–424, 1999.