

# Design and Implementation of an Efficient Priority Queue Data Structure

James Rhodes(✉) and Elise de Doncker

Western Michigan University, Kalamazoo, MI 49008, USA
{james.rhodes,elise.dedoncker}@wmich.edu

**Abstract.** Priority queues are among the most useful of all data structures. Existing priority queues have a vast amount of overhead associated with them. There is a need to have a simple data structure that can be used as a priority queue with low overhead. The data structure should have the operation where the data item with the minimum/maximum value is the next item to be deleted. The data structure should also support the function of a calendar queue where elements with the same or similar priority have the same key. For example, all of today's appointments will have today's date as their key. To that end, a bucket data structure has been developed that has both of these features. We address the functionality and efficiency of the data structure for the applications of adaptive multivariate integration and the 15-puzzle. In adaptive multivariate integration, the key is an error estimate, which is a floating-point number. The data for this application has many items with similar keys and the maximum is the next item to be deleted. The key for the 15-puzzle is generated by a heuristic cost function. The data for this application has many items with the same key and the minimum is the next item to be deleted. This paper presents an implementation of the bucket priority queue and discusses its performance.

**Keywords:** Priority queue · Bucket data structure · Heap · Performance

## 1 Introduction and Background

Various software has been developed to perform multivariate integration of difficult functions using adaptive region partitioning [1,2,5,7,8,13,14,18]. The idea is to reduce the error estimate below a tolerated error. Recent research has been conducted on how to perform multivariate integration efficiently using GPUs [9–12,16,17]. The software utilizes a max heap to hold regions for possible further subdivision. The region with the maximum error is deleted from the max heap and subdivided into smaller regions. The smaller regions are inserted into the max heap, and this process continues until either the total error is sufficiently reduced or a maximum number of function evaluations have been performed. We developed a more efficient data structure to replace the max heap as a priority queue that improves the performance of the application.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13378, pp. 343–357, 2022.
https://doi.org/10.1007/978-3-031-10562-3_25

Heaps are a great data structure but they require constant data manipulation in order to maintain the heap property. The heap property guarantees that the children keys of any node are not less or greater than the parent key for a min or max heap, respectively. In order for heaps to satisfy this property, maintenance is required when both inserting and deleting data. We refer to this process as "heapifying". The objective of our data structure is to improve the CPU time by eliminating the maintenance associated with min/max heaps.

### 1.1 Problem Statement

A min/max heap has insert and delete methods that both have $\mathcal{O}(\log n)$ time complexity. For most applications this is acceptable. However, for some applications where the number of elements on the heap is huge, the performance of the application can be adversely affected. Our novel priority queue data structure lowers the time complexity for target applications, while maintaining reasonable accuracy.

In particular, task partitioning methods support algorithms maintaining task pools, such as some branch-and-bound methods and adaptive numerical integration. Adaptive region partitioning is also applied to produce fine meshes delineating areas of lighting and sharp shadows for image processing such as in ray tracing and radiosity methods, and likewise for intensive partitioning in the vicinity of difficult function behavior in adaptive integration. Our focus is on global adaptive integration methods where important (high error) regions are selected, subdivided and their children regions processed at each step, until the global error estimate falls below the tolerated error or the maximum number of function evaluations is achieved. Previous methods used a heap or linked list priority queue and deleted one region at a time, which is inefficient as the heap may become extremely large. We have developed a bucket-based data structure that improves efficiency without significantly affecting the accuracy by selecting moderately important regions for a parallel evaluation at each step.

### 1.2 Background

A review of data structures and algorithms was conducted as a preliminary step of our development of a priority queue data structure. The most common data structures associated with priority queues and a wide variety of applications were reviewed. Fast implementations of priority queues are important to many high-performance computing (HPC) applications, and hence optimizing their performance has been a subject of intense research [20]. Although priority queues can be implemented with data structures such as simple arrays and linked lists, heaps are most commonly used. After search trees, heaps are the second most studied data structure [3]. Heaps support the insert, find-min (find-max), and delete-min (delete-max) operations. Heaps that support the minimum key operations are called min-heaps, and heaps that support the maximum key operations are called max-heaps.

The heap data structure was invented by J. W. J. Williams in 1964 for a sorting application [21]. This connection is important since sorting $n$ elements has a complexity of $\mathcal{O}(n \log n)$. Building a min heap with $n$ elements can be done in $\mathcal{O}(n)$ time. Then, $n$ delete-min operations can be performed to sort the elements in non-decreasing order. A delete-min operation can get and delete an element in $\mathcal{O}(\log n)$ time, thus "heapsort" has $\mathcal{O}(n \log n)$ time complexity. Heaps have a tree-based structure. They have the property where the key of a parent node is not smaller (greater) than the key of any of its children nodes for a min-heap (max-heap) [3]. There are different types of heaps with various operations and complexities. The "standard" heap is known as a binary heap based on a binary tree. Our data structure improves the efficiency of this type of heap for applications of interest.

The 15-puzzle application is a special case of the $(n^2 - 1)$-puzzle problem, which has applications to practical relocation problems [6,19]. In this case, $n = 4$ $(4^2 - 1 = 15)$. It consists of a $4 \times 4$ matrix with 15 tiles numbered 1 to 15 and an empty space. In the initial state of the matrix, the tiles are not in numerical order. The idea is to arrange the tiles in numerical order from 1 to 15 starting with the top left corner and going left to right and top to bottom. This state is considered the solution or goal state. A tile can be moved up, down, left, or right depending on its position in the matrix. The tiles are moved in such a way as to work towards the goal state. Some applications utilize branch-and-bound or genetic algorithms.

The term branch-and-bound refers to all state space search methods in which all children of the node being expanded (E-node) are generated before any other node can become the E-node. There are two basic search strategies, BFS and D-search [15]. A BFS state space search is a FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue). In D-search, the exploration of a new node cannot begin until the current E-node is fully explored. A D-search state space search is a LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack). In both FIFO and LIFO branch-and-bound, the selection rule for the next E-node is rigid and blind. It does not give any priority to a node that has the best chance of finding the solution node quickly. We will apply an LC (Least Cost) state space search for the 15-puzzle.

An LC state space search uses a cost function to give priority to a node with the least cost. A node with the least cost becomes the E-node. The idea is that the solution node will be found faster if the node with the least cost is expanded next. Since the node with the least cost value has highest priority, a min-heap is used for the 15-puzzle application. The state space is represented by a tree. For example, Fig. 1 displays part of a state space tree (generated breadth-first) for a 15-puzzle instance.

The cost function [15] for the 15-puzzle application is $c(x) = f(x) + g(x)$ where $f(x) =$ the length of the path from the root of the state space tree to node $x$ and $g(x) =$ the number of tiles that are not in the correct position in the matrix for the state of node $x$.

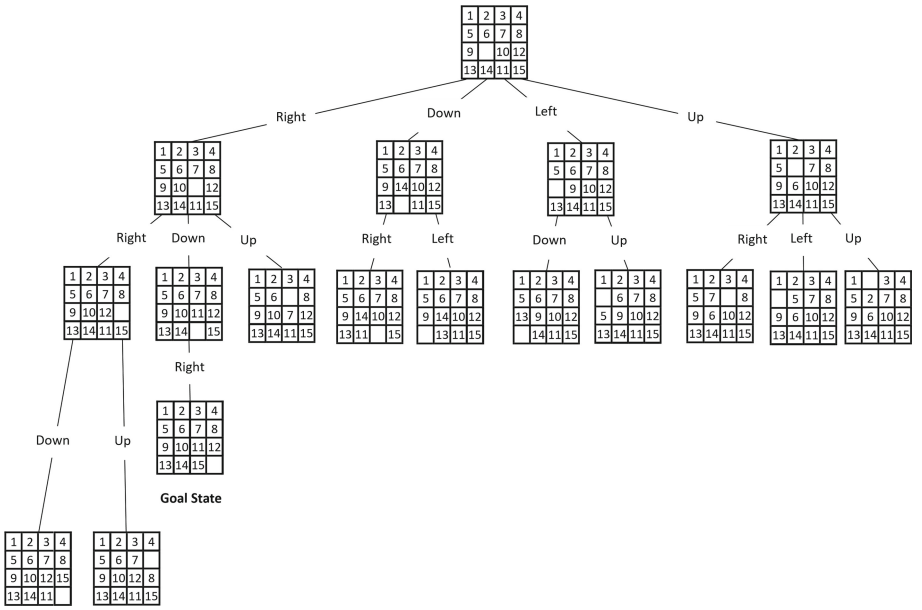


**Fig. 1.** A partial state space tree for the 15-puzzle application

Subsequently, this paper covers methods and implementation in Sect. 2, detailed results in Sect. 3, and gives conclusions in Sect. 4.

## 2 Methodology and Implementation

In order to improve on the efficiency of heaps, the "heapify" process must be optimized or eliminated. Many applications that utilize a heap insert an enormous number of items with the same or similar key value. Each of these items is stored in a separate node. So, it is possible for many nodes with the same or similar key values to be stored on a heap. When a node is deleted from a heap, the "heapify" process rearranges the heap by first moving the last entry (rightmost leaf) of the heap to the root position. The "heapify" function is called recursively and moves the node down through the heap until every node in the heap satisfies the heap property. A similar process occurs when a node is inserted into a heap. If the heap has millions of nodes, this could take up a great deal of CPU time. The smaller the number of nodes on the heap, the less time it takes to "heapify". This leads to the question of whether nodes with the same or similar key values on a heap could be coalesced to make the application more efficient.

We investigated the answer to this question by using buckets. We start this discussion by associating buckets with nodes on a heap; therefore, it is necessary to make a distinction between a data item (or list item) and a heap item. A list item is an individual node that is a member of a linked list. A heap item is the head node of a linked list, corresponding to a heap node that can be inserted into

or deleted from the heap, thus each node on the heap corresponds to a bucket of nodes with the same or similar key value. The bucket may have many nodes in it, organized as a linked list, but only the head node of the linked list will reside on the heap. More precisely, the node on the heap will be a pointer to the head node of a linked list. This means that the node collection could have millions of items in it but less than a percentage of those nodes would have to be manipulated whenever the "heapify" process is called. The nodes in a bucket will be moved as if they were one node.

Also, with buckets, not only is the time spent to "heapify" lowered but the "heapify" process is called less frequently. If there is more than one node on the linked list in the root bucket on a heap, and a list item is deleted from it, at least one node will remain in the linked list, and a call to the "heapify" process will be avoided. Likewise, if a node that is being inserted into the heap has a key value that already exists in a bucket on the heap, a "heapify" process can be avoided by simply placing the new node at the end of the linked list, provided that the correct bucket can be determined.

By using a hashing function on the buckets, this can be done in $\mathcal{O}(1)$ time. This led us to develop a priority queue data structure in the form of a hash table that gets rid of the heap altogether. This not only reduces the number of nodes to manipulate, but eliminates the "heapify" process entirely.

Note that there is a trade-off between the size of the buckets and a possible reduction in accuracy. If, in the multivariate integration application, the regions are not selected in the order of decreasing error estimate, this may degrade the accuracy. For a parallel application it will further be of interest to enable deletion and insertion of multiple regions at a time.

### 2.1 Buckets Implementation

The buckets to store data items will form an array of pointers to the heads of linked lists. The array is dynamic and can grow and shrink as required. If the last node of a bucket is deleted, to find the next (minimum or maximum) bucket, the array is searched linearly to find the next non-null bucket. If the bucket for a new item that is being inserted is empty, there is not any additional data manipulation required.

**Multivariate Integration.** In certain applications, the key values on the linked list will not be required to be exactly the same. For these applications, it is sufficient for the values to be similar or within a range. Multivariate integration is such an application. Previous implementations with a standard heap store an integer index of a region on the heap. The index is used to get a value that represents the error estimate of that region from an array [17]. The error estimate referenced by the index is used as the priority to position the index on the heap. Since the region with the largest error estimate is deleted first, the application utilizes a max heap. Our new data structure stores a node according to its error estimate as well. However, the $log_{10}$ and the mantissa of the value are used

to create a bucket. The first bucket value is at the head of the linked list and a pointer to the head is inserted into the array. Then, when another value is inserted into a bucket with the same $log_{10}$ value and mantissa as a head node of a linked list already in the array, it is placed at the end of the linked list for which that node is the head. If a node with the $log_{10}$ and the mantissa of the value is not in the array, then the value is associated with a new head node and a pointer to the new head node is inserted into the array. The precision (length) of the mantissa is configurable in the data structure. In essence, the bucket structure for the multivariate integration application is a 2-dimensional array of pointers that point to head nodes of linked lists. A hashing function used to find the correct bucket is based on a simple function $g(log_{10}E)$ and the mantissa of the error estimate $E$, for example, BUCKETS[$g(log_{10}E)$][mantissa].

The buckets for the data structure are represented by an array of pointers to the head nodes of the linked lists for the priorities of an application. The priority (error estimate) for the (double precision) multivariate integration application has a non-negative double data type. The desired mantissa length has to be established for bucket purposes. Let's assume the priority queue considers key values with 1 to 6 digits of precision. As mentioned, the BUCKETS array for the multivariate integration application will have two dimensions.

The first dimension represents $log_{10}$ of the error estimate. For example, with the dimension size for the $log_{10}$ index set to 601, the $log_{10}$ value ranges between -300 and 299, inclusively. An offset of 301 is added to the $log_{10}$ value of the error estimate so that the index values will be 1 to 600, inclusively. The index 0 element is used for the value of 0. The second dimension represents the mantissa of the error estimate. The dimension size for the mantissa can be a power of 10 between 10 and 1,000,000, inclusively, depending on the desired mantissa length. If 1 digit of precision is desired, then the dimension size is set to 10. That would allow the index to be in the range from 0 to 9. If 6 digits of precision are desired, then the dimension size is set to 1,000,000. That would allow the index to be a value from 0 to 999,999. So, if 3 digits of precision are desired, then the BUCKETS array is declared as BUCKETS[601][1000]. The size and number of the dimensions may be configured differently depending on the application.

The values of the indexes for the BUCKETS array will be determined by taking the base 10 logarithm of the error estimate, and an offset is added to eliminate negative numbers. This will be the index for the first dimension. The error estimate is multiplied by 10 to the power of $(-)$ its base 10 logarithm plus the base 10 logarithm of the selected precision $-1$. The integer part of this value will provide the mantissa, which is the index for the second dimension. Using $e_a$ to denote the error estimate, and $m_{size}$ for the required mantissa size, the exponent and the mantissa are calculated as follows:

$$\text{exponent } = \lfloor \log_{10}(e_a) \rfloor + \text{offset (to eliminate negative logarithms)}$$

$$\text{mantissa } = \lfloor e_a * 10^{-\lfloor \log_{10}(e_a) \rfloor + \log_{10}(m_{size}) - 1} \rfloor = \lfloor e_a * m_{size} * 10^{-\lfloor \log_{10}(e_a) \rfloor - 1} \rfloor$$

For example, if the error estimate is $e_a = 0.002316$, then the exponent is $\lfloor \log_{10}(0.002316) \rfloor + 301 = -3 + 301 = 298$, and the mantissa with 1 digit of precision is $\lfloor 0.002316 * 10 * 10^{-\lfloor \log_{10}(0.002316) \rfloor - 1} \rfloor = \lfloor 0.002316 * 10 * 10^{-\lfloor -2.635261 \rfloor - 1} \rfloor = \lfloor 0.002316 * 10 * 10^2 \rfloor = \lfloor 2.316 \rfloor = 2$. The mantissa with 3 digits of precision is $\lfloor 0.002316 * 10^3 * 10^{-\lfloor \log_{10}(0.002316) \rfloor - 1} \rfloor = \lfloor 0.002316 * 10^3 * 10^{-\lfloor -2.635261 \rfloor - 1} \rfloor = \lfloor 0.002316 * 10^3 * 10^2 \rfloor = \lfloor 231.6 \rfloor = 231$.

The truncation of the error estimate according to the desired length of precision is only for determining the bucket in which to put the data item. The actual values are stored in the node and used in the application. Thus, the actual error estimates are not affected.

**15-Puzzle.** The 15-puzzle is another heap application that we implemented with our bucket data structure. All key values on the linked list within a bucket will be exactly the same for this application. The (integer) key value represents the cost of a state of the puzzle, which is used as the priority. Since the state with the smallest cost is deleted first, the priority queue implemented as a heap would use a min heap. For the bucket data structure, the cost of a node will be used to create a bucket. The cost of a node equals the number of tiles that are not in their proper place plus the length of the path from the root of the state space tree to the node. The first occurrence of a particular cost becomes the head node of the linked list and a pointer to the head node is inserted into the appropriate bucket. Then, when another cost is inserted with the same value as a head node of a linked list already in the bucket array, it is placed at the end of the linked list for which that node is the head. If no node with the cost value is on the heap, then the value is associated with a new head node and a pointer to the head node is inserted into the array. The buckets for the 15-puzzle application form a 1-dimensional array of pointers that point to the head node of a linked list for a particular cost of a state. A hashing function that is used to find the correct bucket is based on the cost of the node's state, for example, BUCKETS[cost]. Once the hashing function is established, our data structure will operate in the same manner for any application.

The priority (cost) for the 15-puzzle application has an integer data type, is non-negative and the cost is assumed to be less than 100 in the implementation. As mentioned, the BUCKETS array for this application will have one dimension. The dimension size is set to 100, assuming that the cost can be a value between 0 and 99, inclusively. The index for the dimension will also be a value from 0 to 99. The BUCKETS array would be declared as BUCKETS[100].

### 2.2 Further Specifications

There was some consideration of using data structures in the C++ Standard Template Library (STL) to develop the data structure. However, this would force the result to be programming language dependent, which is undesirable. Therefore we used data types common to various programming languages, thus allowing the data structure to be used with any of the popular programming languages.

Our data structure implementation uses an array with one or more dimensions. The data items in the array are pointers to nodes, which are heads of linked lists of nodes. When a data item is inserted, if a node with a certain bucket value is not already present, then the node becomes the head node for that bucket and a pointer to the head node is inserted into the array. If a pointer to a head node with a certain bucket value is already in the array, then the node is added to end of the linked list and it becomes the tail node for that bucket. When a data item is deleted from a linked list, if it is the only node of the linked list, then the pointer to that bucket is set to NULL after deleting the node. If the linked list has more than one node, then this operation is not required.

Our implementation has several methods allowing it to be a self-contained and fully functional data structure that can perform all the operations necessary to accommodate an application. This includes performing its own garbage collection when applicable.

### 2.3 Priority

The priority of a data item that is inserted or deleted by an application is used to determine the bucket in which to put the data item. The priority is also used for comparison purposes to determine the min or max value. This will be different for every application. Our data structure has a MIN pointer that points to the bucket with the minimum value or a MAX pointer that points to the bucket with the maximum value dependent on the application. The priority is the first most important update that has to be made when incorporating the data structure into an application. For the multivariate integration application, the priority is the value of a region's error estimate. The largest value has the highest priority. For the 15-puzzle, the priority is the value of the cost of the state associated with a node. The smallest value has the highest priority. For other applications, someone at the domain or subject matter expert level will have to determine the correct priority.

### 2.4 Hashing

The hashing function to determine the proper bucket for a given priority will be different for every application. This is the second most important update that has to be made when incorporating the data structure into an application. The hashing function has to return a unique index for each priority value. For the multivariate integration application, the hashing function determines the bucket according to the $log_{10}$ and the mantissa of the error estimate with the desired number of digits. For the 15-puzzle application, the hashing function produces the cost of a node's state. For other applications, someone at the domain or subject matter expert level will have to implement a suitable hashing function.

### 2.5 Configuration

The data structure can be configured to function like a min heap or a max heap. It can also be configured to delete one or more data items at a time when the

delete method is called. Some of the other configurable features include settings for the maximum number of data items, the bucket size and precision of the keys in the buckets.

### 2.6 Adjustments

The size of the buckets array can be adjusted according to the number of data items being stored in the array, using methods described in [4] – by allowing it "to oscillate between the top and the bottom of a larger array". For example, "shrinking" can be accomplished by manipulating a subarray. The structure starts out with a default size. Each time the current size is reached, the size of the array will be doubled. If ever the size drops significantly, the current size will be cut in half. The point at which the size of the array is cut in half will be when the size of the data falls below the current size divided by four. However, the array is never reduced to a size lower than the original size.

### 2.7 Required Changes for Different Applications

Our data structure was developed so that the code modifications required for one application or another are minimal. The first step to update the data structure for a particular application is to decide on the configuration settings discussed in the Configuration Sect. 2.5. This includes determining if the application will function as a min or a max heap; if it will delete one or more data items at a time; setting the maximum number of data items, and the size and precision of the buckets. The next step is to implement the "node" class, which encapsulates the type of information that will be inserted and deleted. This will be unique and designed specifically for each application. The only code changes required for the available methods concern the logic necessary for comparing items for priority purposes, the logic for the hash function to determine the bucket in which each data item will be placed, and the actions to take when there is an overflow or underflow. The logic for these code changes will affect the insert, delete, and contains methods. The logic for comparison and the hash function for buckets is isolated in the get_priority and get_bucket methods, respectively. It was our goal to avoid having to change any of the other methods.

## 3 Testing and Results

For this paper, two applications were implemented, multivariate integration and the 15-puzzle. For both of these applications, the behavior of the heap was observed. Normally, the multivariate integration application uses a max heap, and the 15-puzzle uses a min heap. The basic operations of a heap are insert and delete. Whenever an item is inserted into a heap, the position to place the newly added data item has to be determined. This requires a certain amount of CPU time. Likewise, whenever a data item is deleted from a heap, the remaining items have to be rearranged so that the heap property is maintained, which

requires a certain amount of CPU time as well. This "heapify" process is executed recursively. If the application is heap-intensive, meaning millions of data items are inserted into and deleted from the heap, the total time spent inserting and deleting may be substantial.

Testing was conducted on the two applications discussed in this paper, for the 15-puzzle and multivariate integration. The original programs using a heap priority queue for these applications were run. A counter was incremented to keep track of the number of "heapify" calls in the course of each execution.

The programs for these applications were rewritten and the min and max heaps were replaced with the bucket data structure. The programs incorporating the new data structure for the applications were run and the results were obtained. The total time spent in insertions and deletions were recorded and compared to the corresponding times incurred in the heap code.

### 3.1 15-Puzzle

The 15-puzzle application was executed with the initial state shown in Fig. 2, on a x86_64 machine running a Ubuntu version of Linux.

| | | | |
|---|---|---|---|
| 2 | | 8 | 3 |
| 1 | 5 | 6 | 12 |
| 13 | 15 | 9 | 7 |
| 14 | 11 | 10 | 4 |

**Fig. 2.** The initial state used for the 15-puzzle instance

The 15-puzzle heap implementation created a heap with 17,413,988 individual nodes. For example, it executed 8,231,156 direct "heapify" calls when deleting nodes from the heap, which further made 141,475,164 recursive calls for a total 149,706,320 calls. The buckets implementation had the 17,413,988 nodes distributed among 21 buckets.

- For the total deletion time, the heap code took 5.82 s (seconds); the buckets code took 0.71 s for a 87.9% improvement.
- For the total insertion time, the heap code took 2.97 s; the buckets code took 1.23 s, a 58.8% improvement.
- The total insertion and deletion time was 8.79 s for the heap code, and 1.93 s for the buckets code, a 78.0% improvement.
- The total execution time for the puzzle instance was 22.10 s for the heap code, and 13.82 s for the buckets code, a 37.5% improvement.

The number of "heapify" calls is thus extremely large (in the original 15-puzzle application code). The bucket data structure eliminates this type of data manipulation, which results in a substantial reduction in the amount of CPU utilization.

As an added bonus, the original program for the 15-puzzle application found the solution in 8,231,156 iterations. The program rewritten to incorporate the bucket data structure found the solution in 7,174,751 iterations, which is 1,056,405 less iterations than the original program.

For the multivariate integration application, we used adaptive integration code from [17], which performs its region subdivisions sequentially, as well as its heap manipulations, but evaluates the integration rules on the GPU (for two or four regions at a time). A new program was generated by replacing the max heap in the original code by the new bucket data structure.

### 3.2 Multivariate Integration

The multivariate integration application was executed for the test cases in [17], including an integral of a function with discontinuous derivatives, one with a singular integrand, and a linear model function. We observed a similar behavior for the different test cases, and thus report on one of these, shown as the integral $I_1$ below, where $d$ denotes the dimension (set to $d = 10$), and $\mathcal{C}_d$ is the $d$-dimensional unit cube. The exact result of the integral is 1.

$$I_1 = \int_{\mathcal{C}_d} f(\mathbf{x})\ d\mathbf{x} = \frac{6}{5d} \int_{\mathcal{C}_d} \sum_{i=1}^{d} |3x_i - 1|\ d\mathbf{x}. \tag{1}$$

The error tolerance was set to 0, and the maximum number of function evaluations was 5,000,000,000. The original (CUDA) program creates a heap (on the host) with 3,920,800 individual nodes. The integrand evaluations are performed on the GPU. The program incorporating the bucket data structure for the application had the 3,920,800 nodes distributed among 124 buckets, and uses one digit of precision for the length of the mantissa to determine the proper bucket for insertion. The program runs are done on a system with dual 8C Intel Xeon E5-2670 @2.6 GHz as the host, with 128 GB of RAM, and Kepler 20(m) GPU. The Kepler 20(m) GPU has 2496 CUDA cores, 4.8 GB global memory and 956.8 GFLOPS double precision theoretical performance. This is an old GPU with low performance compared to, e.g., the GV100 at 8.33 TFLOPS for FP64 (double).

The original multivariate integration program had 980,199 direct "heapify" calls when deleting nodes from the heap, and 18,468,590 recursive calls for a total of 19,448,789 calls. Albeit a large number, it is only 13.0% of the corresponding number for the 15-puzzle application.

For a time comparison between the heap and buckets versions, we have the following:

**Table 1.** $I_2, I_3$ parameters $\alpha$, integral approximation $Q$, total absolute error $E_a$, processing time improvement (%), and total processing time (inserting and deleting) in seconds, for the heap and buckets data structures

| Int. | $\alpha$ | $Q$ | $E_a$ | | % Improv. | Time [s] | |
|---|---|---|---|---|---|---|---|
| | | | Heap | Buckets | | Heap | Buckets |
| $I_2$ | 2 | 4.483234514 e−02 | 1.44 e−08 | 1.44 e−08 | 36.1 | 6.28 | 4.01 |
| | 4 | 2.42941120 e−03 | 1.41 e−08 | 1.41 e−08 | 40.8 | 6.83 | 4.04 |
| | 8 | 1.94867 e−05 | 5.70 e−09 | 5.58 e−09 | 38.1 | 6.72 | 4.16 |
| | 9 | 1.0053 e−05 | 7.80 e−09 | 7.65 e−09 | 35.4 | 6.31 | 4.08 |
| | 9.2 | 9.6524 e−06 | 9.41 e−09 | 9.24 e−09 | 36.0 | 6.60 | 4.22 |
| | 9.4 | 9.91012 e−06 | 1.28 e−08 | 1.26 e−08 | 43.1 | 7.43 | 4.23 |
| | 9.5 | 1.04454 e−05 | 1.57 e−08 | 1.54 e−08 | 38.9 | 6.90 | 4.21 |
| $I_3$ | 0.1 | 2.8660 e 00 | 1.07 e−02 | 1.09 e−02 | 40.8 | 7.08 | 4.20 |
| | 0.2 | 9.276 e 00 | 1.23 e−01 | 1.23 e−01 | 37.3 | 6.69 | 4.19 |
| | 0.3 | 34.87 e 00 | 1.04 e 00 | 1.05 e 00 | 43.8 | 7.13 | 4.01 |

- The total deletion time was 3.80 s for the heap code, and 0.88s for the buckets version, a 76.9% improvement.
- The total insertion time was 3.44 s for the heap code, vs. 3.25 s for the buckets version, a 5.4% improvement.
- Thus, the total insertion and deletion time was 7.24 s for the heap, and 4.13 s for the buckets code, a 42.9% improvement.
- In this case, however, the total execution time was heavily dominated by the function evaluation time which was 85.0% of the total execution time and amounted to 142.30 s using the heap, vs. 138.13 s using buckets, 2.9% improvement. The data structure performance would be more significant if, for example, a faster GPU would be used effectively.

The test results further demonstrate that the programs rewritten to incorporate the bucket data structure maintained the accuracy for both applications. The original integration program returns the value 1.0000564607886810 e+00 as the total integral approximation, i.e., the sum of the results of all subregions on the heap, with total absolute error estimate 8.334 e−05 as the sum of the absolute errors of all subregions on the heap. These values are very close to the total integral approximation of the program rewritten to incorporate the bucket data structure, 1.0000561894650961 e+00, which is the sum of the results of all subregions, with 8.294 e−05 as the sum of the absolute errors of all subregions.

Thus, the accuracy of the multivariate integration application was not affected.

As additional test problems, we consider the integrals

$$I_2 = \int_{C_d} \frac{1}{(\sum_{i=1}^{d} x_i)^{\alpha}} \, d\mathbf{x}, \quad \alpha < 10 \tag{2}$$

**Table 2.** $I_2, I_3$ parameters $\alpha$, the number of "heapify" calls on insert, on delete (directly, recursively, and the total), and the number of buckets required for the heap and buckets data structures

| Int. | $\alpha$ | Heapify/Ins. | Heapify/Del. | Rec. Heapify | Tot. Heapify/Del. | Buckets |
|---|---|---|---|---|---|---|
| $I_2$ | 2 | 3,920,800 | 980,199 | 18,677,644 | 19,657,843 | 85 |
| | 4 | 3,920,800 | 980,199 | 18,655,439 | 19,635,638 | 90 |
| | 8 | 3,920,800 | 980,199 | 18,728,412 | 19,708,611 | 110 |
| | 9 | 3,920,800 | 980,199 | 18,744,088 | 19,724,287 | 164 |
| | 9.2 | 3,920,800 | 980,199 | 18,767,907 | 19,748,106 | 191 |
| | 9.4 | 3,920,800 | 980,199 | 18,770,462 | 19,750,661 | 241 |
| | 9.5 | 3,920,800 | 980,199 | 18,781,981 | 19,762,180 | 287 |
| $I_3$ | 0.1 | 3,920,800 | 980,199 | 18,641,266 | 19,621,465 | 82 |
| | 0.2 | 3,920,800 | 980,199 | 18,686,978 | 19,667,177 | 86 |
| | 0.3 | 3,920,800 | 980,199 | 18,896,512 | 19,876,711 | 83 |

and

$$I_3 = \int_{\mathcal{C}_d} \frac{1}{\prod_{i=1}^{d} x_i^{\alpha}} \, d\mathbf{x}, \quad \alpha < 1, \tag{3}$$

both for $d = 10$ dimensions. $I_2$ has an integrand singularity at the origin, and the integrand function of $I_3$ is singular at $x_i = 0$ for $1 \leq i \leq d$, which will cause excessive subdivisions by the adaptive strategy near the singularities.

Table 1 displays total loads (in terms of error), the percentage of processing time improvement that the buckets data structure provided, and processing times (inserting and deleting) for the heap and buckets data structures.

The efficiency of the buckets data structure also emerges from the Table 2 data, showing the number of "heapify" calls when inserting, deleting (directly, recursively, and the total) in the heap code, and the number of buckets that were required to accommodate all the nodes that were inserted and deleted in the buckets code. The number of "heapify" calls when inserting (3,920,800) corresponds to the number of regions generated, and the number of "heapify" calls when deleting directly (980,199) corresponds to the number of regions evaluated. The number of regions generated is four times the number of regions evaluated since, for every region evaluated, four subregions are generated.

## 4 Conclusions

We presented a priority queue data structure based on buckets to group the same or similar data items together, which makes the data structure more efficient than a heap for the applications at hand. A heap inserts these data items as separate nodes, while using the "heapify" process to maintain the heap property. In the bucket data structure, data items with the same or similar values reside

on linked lists within the buckets and can be inserted or deleted without relying on "heapify" calls. In essence, this eliminates the need for a heap in the applications for multivariate integration and the $(n^2 - 1)$-puzzle/relocation problem considered. The only cost is the time to find the bucket in which to put a data item, an operation that has a $\mathcal{O}(1)$ time complexity using a hashing function. The actual values of the data items are stored, despite the fact that a truncated value may be used in the hashing function to determine the bucket. This allows the data structure to be more efficient than a min/max heap without affecting the accuracy of the results.

Our data structure can be used with any application that uses a min/max heap. It can be reconfigured for a different application with minimal effort. The most difficult part of configuring the data structure for a different application is determining the priority of the data items and the hashing function for the buckets. This will have to be implemented by someone who is familiar with the data at the domain or subject matter expert level. In future work we will continue to test with other applications, as priority queues constitute one of the most utilized data structures in Computer and Computational Sciences.

**Acknowledgements.** The authors would like to thank Dr. J. Kapenga and Dr. D. Zeitler for their valuable comments, as well as the reviewers of this paper.

## References

1. Berntsen, J., Espelid, T.O., Genz, A.: An adaptive algorithm for the approximate calculation of multiple integrals. ACM Trans. Math. Softw. **17**, 437–451 (1991)
2. Berntsen, J., Espelid, T.O., Genz, A.: Algorithm 698: DCUHRE-an adaptive multidimensional integration routine for a vector of integrals. ACM Trans. Math. Softw. **17**, 452–456 (1991)
3. Brass, P.: Advanced Data Structures. Cambridge University Press, Cambridge (2008)
4. Brown, R.: Calendar queues: a fast o(1) priority queue implementation for the simulation event set problem. Commun. ACM **31**(10), 1220–1227 (1988)
5. Cools, R., Haegemans, A.: CUBPACK: progress report. In: Espelid, T.O., Genz, A.C. (eds.) Numerical Integration, Recent Developments, Software and Applications, pp. 305–315. NATO ASI Series C: Mathematical and Physical Sciences (1992)
6. Demaine, E.D., Rudoy, M.: A simple proof that the $(n^2 - 1)$-puzzle is hard (2017)
7. de Doncker, E., Genz, A., Gupta, A., Zanny, R.: Tools for distributed adaptive multivariate integration on NOW's: PARINT1.0 release. In: Supercomputing'98 (1998)
8. de Doncker, E., Kaugars, K., Cucos, L., Zanny, R.: Current status of the ParInt package for parallel multivariate integration. In: Proceedings of Computational Particle Physics Symposium (CPP 2001), pp. 110–119 (2001)
9. de Doncker, E., Yuasa, F.: Self-energy Feynman diagrams with four loops and 11 internal lines. In: Gervasi, O., et al. (eds.) ICCSA 2021. LNCS, vol. 12953, pp. 160–175. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86976-2_11
10. de Doncker, E., Yuasa, F., Almulihi, A.: Efficient GPU Integration for Multi-loop Feynman Diagrams with Massless Internal Lines. In: Okada, H., Atluri, S.N. (eds.) ICCES 2019. MMS, vol. 75, pp. 737–747. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27053-7_62

11. de Doncker, E., Yuasa, F., Almulihi, A., Nakasato, N., Daisaka, H., Ishikawa, T.: Numerical multi-loop integration on heterogeneous many-core processors. In: The Journal of Physics: Conference Series (JPCS), vol. 1525, no. 012002 (2019). https://doi.org/10.1088/1742-6596/1525/1/012002
12. de Doncker, E., Yuasa, F., Olagbemi, O., Ishikawa, T.: Large scale automatic computations for Feynman diagrams with up to five loops. In: Gervasi, O., et al. (eds.) ICCSA 2020. LNCS, vol. 12253, pp. 145–162. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58814-4_11
13. Genz, A., Malik, A.: An adaptive algorithm for numerical integration over an n-dimensional rectangular region. J. Comput. Appl. Math. **6**, 295–302 (1980)
14. Hahn, T.: Cuba − a library for multidimensional numerical integration. Comput. Phys. Commun. **176**, 712–713 (2007)
15. Horowitz, E., Sahni, S., Rajasekaran, B.: Computer Algorithms/C++. Computer Science Press (1997)
16. Jarząbek, Ł, Czarnul, P.: Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. J. Supercomput. **73**(12), 5378–5401 (2017). https://doi.org/10.1007/s11227-017-2091-x
17. Olagbemi, O.E., de Doncker, E.: Scalable algorithms for multivariate integration with ParAdapt and CUDA. In: Proceedings of the 2019 International Conference on Computer Science and Computational Intelligence. IEEE Computer Society (2019)
18. Piessens, R., de Doncker, E., Überhuber, C.W., Kahaner, D.K.: QUADPACK, A Subroutine Package for Automatic Integration, Springer Series in Computational Mathematics, vol. 1. Springer, Cham (1983). https://doi.org/10.1007/978-3-642-61786-7
19. Ratner, D., Warmuth, M.: The $(n^2 - 1)$-puzzle and related relocation problems. J. Symb. Comput. **10**(2), 111–137 (1990)
20. Ros-Giralt, J., Commike, A., Cullen, P., Lucovsky, J., Madathil, D., Lethin, R.: Multiresolution priority queues (2017)
21. Williams, J.W.J.: Algorithm 232: heapsort. Commun. ACM **7**(6), 374–378 (1964)