Discrete Range Searching Primitive for the GPU and Its Applications

JYOTHISH SOMAN, IBM India Research Labs
KISHORE KOTHAPALLI and P. J. NARAYANAN, International Institute of Information Technology, Hyderabad

Graphics processing units (GPUs) provide large computational power at a very low price, which position GPUs well as an ubiquitous accelerator. However, GPUs are space constrained, and hence applications developed for GPUs are space sensitive.

Space-constrained computational devices such as GPUs can greatly benefit from representations that reduce space consumption drastically. One such representation is the succinct representation of trees. Succinct representation of trees generally allows for operations such as parent queries, least common ancestor queries, and so on. Mapping such a robust representation to the GPU for targeted applications can lead to substantial improvement in problem sizes that are processed at a given point of time. Space-saving methods such as succinct data structures remain largely unexplored on the GPU. In this work, a succinct representation of ordered trees on the GPU is explored, with application to discrete range searching (DRS).

Based on the succinct representations found applicable, a space–saving solution for DRS is presented here. In our method, DRS is mapped to a least common ancestor query on a Cartesian tree. For space-efficient DRS queries, we store the succinct representation of the Cartesian tree of an array. Our method uses a maximum of 7.5 bits of additional space per element. Furthermore, the speed-up achieved by our method is in the range of 20–25 for preprocessing and 25–35 for batch querying over a sequential implementation. Compared to an 8-threaded implementation, our preprocessing and querying methods obtain a speed-up of 6–8.

We also study the applications of the DRS on the GPU. Efficient primitives expand the range of applications performed on the GPU. DRS is one such primitive with direct applications to string processing, document and text retrieval systems, and least common ancestor queries. We suggest that graph algorithms that use the least common ancestor, can be enabled on the GPU based on DRS primitive. We also show some applications of DRS in tree queries and string querying.

Categories and Subject Descriptors: D.1.3 [**GPGPU Programming and Algorithms**]: Concurrent Programming

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: GPGPU, succinct representation, tree queries, range searching

ACM Reference Format:
Soman, J., Kothapalli, K., and Narayanan, P. J. 2012. Discrete range searching primitive for the GPU and its applications. ACM J. Exp. Algor. 17, 4, Article 4.5 (October 2012), 17 pages.
DOI = 10.1145/2133803.2345679 http://doi.acm.org/10.1145/2133803.2345679

1. INTRODUCTION

Graphics processing unit (GPU)-assisted acceleration has been quickly considered to be a valid high-performance computing method. A detailed description of the GPU

Authors' addresses: J. Soman, IBM Research, India 4 Block C, Institutional Area, Vasant Kunj, New Delhi 110070, India (Part of this work was done while at International Institute of Information Technology, Hyderabad); K. Kishore, Center for Security, Theory and Algorithm Research; P J Narayanan, Center for Visual Information Technology International Institute of Information Technology, Hyderabad, India.

DOI 10.1145/2133803.2345679 http://doi.acm.org/10.1145/2133803.2345679

architecture can be found in Nvidia [2007]. GPUs are suited for fine-grained data-parallel applications. Application and algorithm engineering for a GPU is different from that of the CPU. To accommodate architectural level nuances, GPU requires programming models different from the CPU models. Current programming models for GPUs include CUDA [Nvidia 2007], STREAM [Bayoumi et al. 2009], and OpenCL [Stone et al. 2010].

Computing, in general, is dependent on a set of basic algorithms and operations. These set of algorithms have been named as motifs or dwarfs [Asanovic et al. 2006]. GPUs have been an intermittent compute model. The novel compute model of the GPUs has little history of general-purpose algorithmic research. Research on algorithmic motifs has been substantial since the release of first version of CUDA [Nvidia 2007]. Focus has been on designing optimal algorithmic motifs for the GPU. Due to the non-traditional compute model, this has been an important aspect of GPU-centric research. A reduction of programming effort also requires efficient algorithmic motifs. GPU computing, in general, relies on dividing a problem into smaller algorithmic motifs and optimizing these motifs. Such motifs are referred to as primitives in the context of the GPU. Common primitives include `scan` [Sengupta et al. 2007], `sort` [Satish et al. 2009; Sintorn and Assarsson 2008; Cederman and Tsigas 2008], `split` [Patidar and Narayanan 2009; Scheuermann and Hensley 2007], and `list-ranking` [Rehman et al. 2009].

Discrete range searching (DRS) is a yet-unexplored primitive on the GPU, which has wide spread applications in graph theory, string processing, VLSI, document retrieval, and biology [Bentley and Friedman 1979]. To further the claim of DRS as a GPU primitive, in Section 7.1, we present ways in which tree queries on the GPU can be answered efficiently using DRS.

In this article, discrete range search focuses on the range minima query (RMQ) on static arrays. Given an array A, with $|A| = n$, a formal definition for RMQ is:

$$RMinQ_A[i, j] = k | k \in [i, j] \text{ and } A[k] \leq A[l], \text{ for all } l \in [i, j], 0 \leq i, j < n$$

Space-saving techniques have found relevance in DRS problems in general. Solutions for DRS have a specific focus on space-saving methods due to the otherwise polynomial space requirement. Succinct tree representations are a well-studied field of research with a large body of theoretical results. Succinct tree representation finds a direct application in the DRS problem, especially for a space-saving design of a solution. In this work, DRS on GPU is explored using succinct representations.

1.1. Related Work

Range minima has been found to form a dual with the least common ancestor (LCA). In the first work on this subject, Gabow et al. [1984] suggest that the minima given a range in an array is equivalent to finding the least common ancestor on the Cartesian tree of the array (Cartesian tree is a heap ordered tree representation of an array). The LCA problem is well studied, and multiple solutions exist for the LCA problem.

A parallel algorithm for the range minima problem was first suggested by Berkman and Vishkin [1989]. Berkman and Vishkin suggest that LCA on a Cartesian tree can be found by finding the range minima on an array containing the heights of elements arranged according to the Euler tour of the tree. The heights in adjacent elements in the array differ by 1; hence, such an array is called as a ± 1 array. On this array, range minima is equivalent to range minima on the main array. RMQ queries can be answered in $O(1)$ time on the ± 1 array. The bookkeeping required substantially increases the space used. It is to be noted that our succinct representation performs a similar mapping of an array, with an intermediate Cartesian tree, to a ± 1 array. Our method does not require any additional space and is fully self-sufficient for querying.

Table I. Space Complexity of Various Succinct Methods for RMQ

Method	Final Space
Berkman and Vishkin [1989]	$O(n \log n) + \|A\|$
Bender et al. [2001]	$O(n \log n) + \|A\|$
Fischer and Heun [2007]	$2n + o(n) + \|A\|$
Fischer et al. [2008]	$O(nH_k) + o(n) + \|A\|$
Sadakane [2007]	$2n + o(n) + \|A\|$
`Our Method`	$2n + o(n) + \|A\|$

A simple and practical algorithm for the RMQ problem is presented by Bender et al. [2001]. Bender et al. present a $\Theta(n \log n)$ space and $\Theta(1)$ time algorithm to solve the range minima problem. Given an array A, $|A| = n$, the array A is preprocessed to form a look-up table M, such that $M[i][j]$, $i \in [0, n-1]$, $j \in [1, \log n]$ contains the index of the smallest element in the range $[i, i + 2^{j-1} - 1]$. Any query for the range $[i, j]$ can thus be answered as follows.

$$RMQ(i, j) = \arg\min\{M[i][k], M[j-t-1][k]\} : t = 2^k,\ t < j - i < 2t$$

Bender et al. [2001] further present a look-up table–based hierarchical method that uses $O(n)$ space and $O(1)$ time per query.

Fischer and Heun [2006, 2007] present a succinct representation-based solution. They use a succinct representation for the lower levels of the hierarchy of the method presented by Bender et al. [2001]. They represent an array by its Cartesian tree and store the Cartesian rank of the tree. In their method, all possible queries on all possible Cartesian ranks are stored by their method. The space required to store each array is smaller than that of Bender et al., but the look-up table size is comparatively large. The technique presented in Fischer and Heun [2006] has comparable per query time with other succinct representation based methods in literature. In our method, we have used a succinct representation of the Cartesian tree. The look-up table we utilized is independent of the size of the array used to form the Cartesian tree. A GPU-centric implementation of range query is presented in our earlier work [Soman et al. 2010].

1.2. Our Contributions

We have presented the first implementation of succinct representation on the GPU. The RMQ algorithm developed has been found to be efficient on the GPU. Also, we were able to achieve speed-ups of over 6–8 over a multithreaded CPU implementation. The method was hence used to support further applications for the GPU. It is our claim that RMQ can be considered as a primitive on the GPU. We further state that given the space constrained computational model of the GPU, succinct representations are worth exploring.

Our contributions can be summarized as follows:

—exploration of succinct tree representation for the GPU computational model;
—bridging ideas from succinct tree literature and range minima query to form an robust RMQ algorithm;
—an efficient RMQ primitive designed for the GPU computational model;
—with minimal space trade-off, fastest range query results to date;
—applications of RMQ, especially in the context of GPU computing.

The rest of this article is arranged as follows: Section 2 presents the generic framework that the RMQs employ. Section 3 presents the challenges of implementing the RMQ framework on the GPU. Section 4 presents the detailed explanation of the succinct

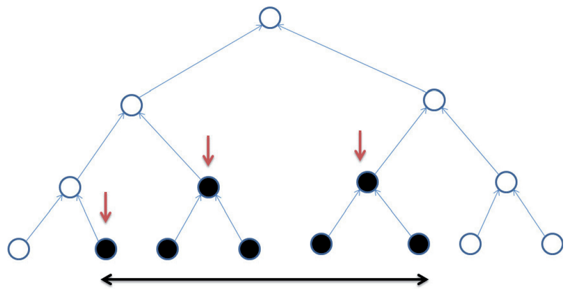


Fig. 1. A binary query tree showing decomposition of a query into sub queries.

representation we have employed. Section 5 presents our method. Section 6 presents the implementation level details and the results of the experiments. Section 7 presents the applications of the RMQ algorithm to multiple methods.

2. RANGE QUERY FRAMEWORK

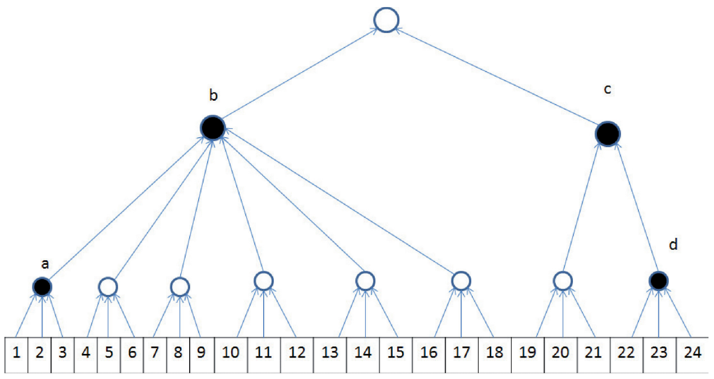
Bender et al. [2001] presented a hierarchical method to perform range queries. Fischer and Heun [2007] improved the space requirement for the same in their hierarchical method. Their space utilization for only the representation is less than a byte per element of the array. Hence, the space requirement of the Fischer-Heun method is approximately $8n$ bits. Although both methods show a limited depth in their hierarchy, a generalization of the hierarchical schema of the two methods follows here.

Their methods can be seen as an analogy to a query tree. The preprocessing algorithm can be interpreted as the formation of a query tree, with each node storing the minimum of all the leaf nodes in its sub tree. Additionally, each inner node of the query tree stores enough information to find the minimum of any pair of its child nodes. The number of children for a node is variable, depending on the height of the node. For a given height, the number of children is kept constant. A binary tree equivalent is shown in Figure 1, where each node stores only the minimum of its two children. A query (2,6) is shown, and inner nodes that inside the query are shown in black. The arrows point to nodes that have sufficient information to perform the query.

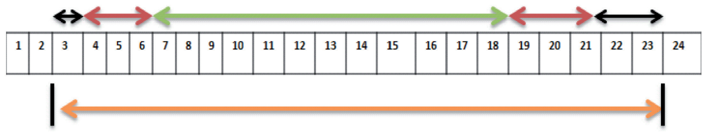
For generalizing to k-ary trees, each inner node contains sufficient information to answer queries for any pair of children of the node. Querying on such a structure involves finding the correct set of inner nodes that can answer the query such that the number of such nodes is minimum. For this, a query decomposition technique is utilized. Each query node falls on the shortest path between the two nodes being queried. The nodes on which the query is performed should have at least two child nodes in the given range.

In their method, a given query is divided into smaller queries, and the partial results are combined together to find the complete result. For example, let the query tree consist of three levels, the last two of fixed size b_2, b_1. For a query pair $\langle i, j\rangle$, $i = i_1 \cdot b_2 . b_1 - i_2 \cdot b_1 - i_3$, and $j = j_1 \cdot b_2 . b_1 + j_2 \cdot b_1 + j_3$. The range minima query from i to j is equal to the minima of all the query pairs.

q_1	i	$i_1 \cdot b_2 . b_1 - i_2 \cdot b_1$
q_2	$i_1 \cdot b_2 . b_1 - i_2 \cdot b_1$	$i_1 \cdot b_2 . b_1$
q_3	$i_1 \cdot b_2 . b_1$	$j_1 \cdot b_2 . b_1$
q_4	$j_1 \cdot b_2 . b_1$	$j_1 \cdot b_2 . b_1 + j_2 \cdot b_1$
q_5	$j_1 \cdot b_2 . b_1 + j_2 \cdot b_1$	j



(a) Decomposition of a query on a k-ary tree.



(b) Array representation of query decomposition.

Fig. 2. Decomposition of a query into multiple subqueries. Here, $b2 = 6$, $b1 = 3$, and query = 3,23.

An example of the this is shown in Figure 2. Here, $b_1 = 3, b_2 = 6$. The resulting queries are $\langle 3, 3\rangle$, $\langle 4, 18\rangle$, $\langle 19, 21\rangle$, and $\langle 22, 23\rangle$. The queries can be represented as nodes on the k-ary query tree shown in Figure 2(a), as point a, b, c, and d, respectively.

In our solution, at each level, the size of the array is reduced by a factor of B. The first level is divided into two stages, one of size B_1 and other of size $B_2 = \frac{B}{B_1}$. It can be noted here that the first level is divided into two stages to reduce space utilization, as the first level requires the maximum space. At the first level, the first stage uses a space-optimal succinct representation to reduce space requirements. Here, we first represent an array by its Cartesian tree. This tree is then represented by its succinct representation. This reduces the space requirement to store the tree. The second stage uses a comparitively space-consuming but faster succinct representation.

It is notable that the look-up table in both algorithms (Fischer and Heun [2007] and Bender et al. [2001]) takes up substantial space and is required by each query. For a CPU-centric method, with a user-invisible cache heirarchy, such a method is practical. On a GPU, the memory heirarchy is fully user pragrammable and needs careful user level optimizations. A more robust succinct representation, requiring a minimal look-up table, is more suitable. In the following section, we present our succint representation, which does not require a large look-up table. A space-computation trade-off is done, maintaining the constant time requirement for each query.

3. CHALLENGES OF RANGE QUERY ON THE GPU

Implementing any algorithm on the GPU requires sensitivity to the architecture and compute model of the GPU. The following challenges are found while implementing RMQ on the GPU.

(1) Limited on chip memory available, as well as limited global memory accessible for the GPU.
(2) The number of registers available on the GPU per thread is limited; 16,384 registers are available per thread block on a Tesla C1060. If the total register requirement

per thread block exceeds 16,384, then the register allotment overflows into other available memory. This causes performance degradation. Reducing the number of threads per thread block to increase the register count per thread increases the time for the algorithms.

(3) Thread divergence is another factor that decides the performance on the GPU. Algorithms that cause a large number of divergent branches to be present at a given point of time per warp cause the performance to degrade.

(4) Large look-up tables require efficient methods to manage data, making use of large RAM and file system. GPU computing introduces a large overhead to access CPU-assisted resources. Hence, space-efficient look-up tables are also necessary.

4. A SUCCINCT REPRESENTATION OF A CARTESIAN TREE

A tree can be reconstructed given two different types of traversals of the same. For succinct representations, the given tree is converted to another tree such that, for a given tree size, any one of the traversals can regenerate the tree irrespective of the structure or topology of the tree. Hence, the traversal is stored succinctly. One such transformation is done by assigning a value of one to each node, and zero to every absent child of each node. Let such a tree be called the *padded tree*. An important property of a padded tree is that the inorder traversal of such a tree will always be a string of alternate zeros and ones. Hence, any other traversal of the tree is sufficient to generate the tree.

Some interesting properties of the padded tree include the following.

—The number of zeros in the subtree rooted at the given node is always greater than the number of ones by one.
—The enumeration of the leaf nodes in preorder, postorder, and inorder traversal of the tree are the same.
—For a Cartesian tree, the i^{th} zero represents the i^{th} element in the base array.
—In the preorder traversal of a padded Cartesian tree, for the i^{th} zero, let the number of ones on the left be Ones(i). Then, Ones(i)-i represents the number of left parents the i^{th} node has. Left parents of i denote parent nodes for which have i in their left subtree.
—In the preorder traversal, if $i = lca(a, b)$, then the i^{th} zero will always be between the a^{th} and the b^{th} zero.

Based on the properties of a padded tree, it is possible to represent any tree as a string of size $2n$ bits. If any other traversal of the tree is available other than the inorder traversal, the original structure of the tree can be reconstructed. Designing a succinct representation is hence converted into finding an optimal traversal of the tree. The important and relevant traversals are level order, preorder, and postorder. All of these traversals are used for the purpose of succinct representations. Variants based on these and further extensions and optimizations to extend to general trees are also studied. We will now use this property to formulate a strategy for answering LCA queries.

4.1. Answering LCA Queries on a Preorder Traversal of a Padded Tree

As presented in the previous section, if $i = lca(a, b)$, then i^{th} zero will always be between a^{th} and b^{th} zero. $lca(a, b)$ is the highest node with a in the left subtree and b in the right subtree. In preorder representation, as soon as a left tree is closed, a zero is placed in the bit string. A one is placed as soon as th left subtree of a node is explored. Thus, $i = lca(a, b)$ will have its left subtree opened before a opened its left subtree and closed before b opened its left subtree. Also, a should close its left subtree before i.

To find the $lca(a, b)$ in linear time, we substitute each zero in bitstring S with -1, to form S'. From the a^{th} zero to the b^{th} zero, we can perform a prefix sum. There can be multiple minimums; each minimum represents a node that has the least number of left parents between a and b. The first minimum represents the $lca(a, b)$.

Hence, $lca(a, b) = Arg(Minimum(prefix\_sum(S'(a, b))))$.

To find the $lca(a, b)$, we need the following information, position of a^{th} and b^{th} zero in S, and prefix sum of S'. To perform prefix sum, a linear scan is sufficient. To accelerate the process, we preprocess the prefix sum value for each bit string of size sz. If the length of S' is L, then the prefix sum of S' can be found by concatenating results of L/sz blocks and merging the results. This method is a minor extension of parallel prefix sum algorithm [JáJá 1992].

To find the a^{th} and b^{th} zero in S is equivalent to finding the a^{th} and b^{th} -1 in S'. This can again be found from the prefix scan of S'. Hence, the succinct representation presented here is self-sufficient in answering RMQ queries. Additional overheads are related to speeding up the scan process.

5. OUR METHOD

Based on the DFUDS succinct representation of ordered trees presented by Jansson et al. [2007], a Cartesian tree–specific modification suitable for the GPU is presented in Section 4. The range minima algorithm presented here is based on the same representation. The complete preprocessing as well as querying phase is performed on the GPU. The preprocessing as well as the the querying phase are explained in the following sections.

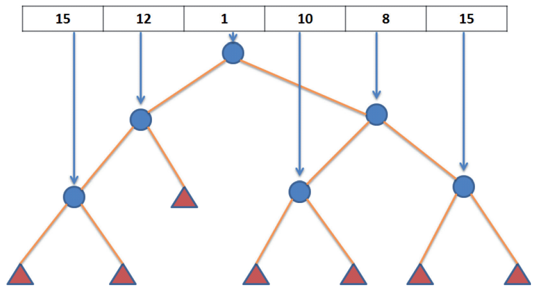
5.1. Preprocessing

Our method is divided into three levels. Each level uses a range minima method which is suited to the data size at the given level. Level 1 focuses on minimizing space, Level 2 focuses on fast querying, and Level 3 focuses on using minimal additional space so as to maximize coalescing while querying. A detailed discussion on these methods follows.

5.1.1. Preprocessing in Level 1. Level 1 is divided into two stages. In Stage 1, the array A is divided into blocks of size B_1. Each block is then represented by a Cartesian tree. The Cartesian tree is stored succinctly. The minimum of each block is stored in an array A'. In Stage 2, the array A' is divided into blocks of size B_2. Array A' is preprocessed to answer each query.

5.1.2. Preprocessing in Stage 1. Jansson et al. [2007] presented a method to represent ordered trees succinctly. Cartesian trees are ordered trees, with the number of children at most two. It has been shown that the representation is able to answer LCA queries without any additional data structures. To accelerate the process, succinct dictionaries are provided that accelerate the process of finding the LCA. The dictionary is independent of the size of the tree. A detailed mapping of the method of Jansson et al. [2007] is shown in Section 4. In our method, we do not add any additional data to the succinct information, but we rely on succinct dictionaries for the purpose of providing a $O(1)$-time solution.

As presented in Section 4, an array can be represented by a succinct representation that stores the preorder traversal of the padded Cartesian tree of the array. The resultant traversal can be stored in a bit string. To generate such a representation, we first generate a padded Cartesian tree for the given array. This is done by performing a nearest smaller values search for each element. A preorder traversal is then performed on this structure. For each block B of size B_1, the preorder ± 1 bit string is formed. The



Pre-order Code:1110001100100

Fig. 3. Conversion of an array into its succinct representation using an intermediate tree representation. Here, blue circles represent actual tree nodes, and red triangle represent padded nodes.

ALGORITHM 1: Preprocessing in Level 1: Stage 1

```
for Each Block $B$ do
  for Each element $e \in B$ do
    Find the nearest smaller values of $e$ to its left and right within $B$
  end for
  Form the Cartesian tree $T$ of $B$ using the nearest smaller values
  Form the padded tree $T'$ from $T$
  Perform a sequential preorder traversal on the padded tree $T'$
  Store the preorder traversal as a bit string
end for
```

detailed algorithm is presented in Algorithm 1. The resultant string of 1 and 0 is then stored. The resulting string requires $2B_1 + 1$ bits of space. Neglecting the trailing zero, the representation uses exactly $2B_1$ bits of space. An example is shown in Figure 3. A brief description of the algorithm is given in Algorithm 1.

5.1.3. Preprocessing for Stage 2. The succinct representation for the first stage works well for small sizes of B_1. For large block sizes, the number of memory and compute operations required to perform a query outweigh the space advantage. Hence, alternative methods, which may be space intensive but can answer queries fast, are used. The method suggested by Bender et al. [2001] is used here. The block size is taken as B_2. For a block of size 2^k, $0 < k < \log(B_2)$; for each element i, the index with the minimum value in the range i to $i + 2^k$ is found, and the relative index with regard to i is stored.

Observe that for an index i, the minimum in a range of size 2^k is equal to the minimum in the range i to $2^{k-1} - 1$ and $i + 2^{k-1}$ to $i + 2^k - 1$. This leads to a natural iterative method. For each element, the resultant relative indexes are packed into limited number of words. Compared to explicitly storing the indexes and minimum values, this method is used to save space. For blocks of size B_2, $\frac{\log(B_2)\cdot\log(B_2-1)}{2}$ bits are required for storing the result.

5.1.4. Preprocessing for Level 2 and Level 3. The data sizes have substantially decreased in this level, and now methods that need not be space saving but can provide faster results while querying are explored for this level. For each query in Level 1, it can be noticed that four global memory reads are required. This is reduced to three global memory reads by following a data-centric method. In Level 2, the size of data has reduced by a factor of $B = B_1 \cdot B_2$. Assuming that the value of B is large enough to reduce the size of the array appreciably, we state that space is no longer a constraint.

ALGORITHM 2: Algorithm to Find RMQ in Level 1: Stage 1 between indices $(i, j), i < j$, given preorder traversal bit string of block containing i and j

```
FindRMQ(i,j,bit_string,BS)
  Divide bit string into blocks of size $BS$
  Find $Block[i]$ and $Block[j]$ containing $i^{th}$ zero and $j^{th}$ zero
  Find bit position ($Bp[i]$) of $i^{th}$ zero in $Block[i]$, and $Bp[j]$, using stored data for sub blocks of size $BS/2$
  Find minimum point(mp) to right of $Bp[i]$ in $Block[i]$
  for Each block $Block[k]$ between $Block[i]$,$Block[j]$ do
    Update $mp$ using bit string of $Block[k]$
  end for
  Update $mp$ using bit string to left of $Bp[j]$+1 in $Block[j]$
  Return zero-count to left of $mp$+1
```

In Level 2, the algorithm of Bender et al. [2001] is used without any succinct representation. Here, the value of the minimum of each block and the corresponding relative index of the minima of each block is stored. It is to be noted that the relative index to the beginning of the block is stored. This assists in saving space. Hence, $\Theta(\log^2(B))$ space is used per element for storing the indexes.

We focus on assigning appropriate block sizes for each level such that Level 3 has a small number of elements. The representation presented in Stage 2 of Level 1 can be reused here. To accelerate querying, the generated representation can be stored in the texture memory. It should be noted that for all the other stages, we use the global memory to maintain preprocessed data for the query phase.

5.2. Querying

The important stage in querying in our method lies in the first stage, which uses the succinct representation. It is to be noted that, the array in this level is represented as a bit array, using an intermediate preorder traversal of a padded Cartesian tree. Querying in this stage is derived from the lca method of Jansson et al. [2007]. Section 4.1 discusses the method in limited detail. For a Cartesian tree of n nodes, $lca(i, j)$ equals $RMQ(i, j)$ for $0 \leq i, j \leq n - 1$. As presented in Section 4.1, RMQ information can be found by linearly parsing through the succinct representation of the Cartesian tree once. To support RMQ queries on the succinct representation of Stage 1 of Level 1, we propose that instead of adding additional data to the bit string (succinct representation) of each block, we can preprocess all bit sequences of a predetermined length. This forms a succinct dictionary that can be accessed for fast querying.

This produces a strong upper bound of 2 bits per node while supporting constant time querying. For n nodes, the length of the succinct representation is $2n$. Let the length of bit strings preprocessed be $BS < n$. For bit strings of size BS, we store the number of zeros in the string and the minimum of the difference between the number of ones and zeros to the left of each position in the string. We do the same for strings of size $BS/2$. It should be noted that $BS/2$ can be any divisor of n. The space taken by the succinct dictionary for block size BS is $2^{BS}log(BS)$. As BS is independent of n, the additional space can be considered as constant. A succinct dictionary saves us from traversing the succinct representation bit by bit and reduces memory operations substantially and reducing divergence. The method is stated in Algorithm 2.

Querying on the GPU for the other levels is trivially based on the preprocessing algorithm. It can be noted the nature of the querying is very similar to that of Bender et al. [2001]. Hence, the discussion on querying in other stages can be followed from Bender et al. [2001].

Table II. Parameters for RMQ: Size of Each Stage/Level and Number of Elements Processed at Each Level/Stage

Stage	Size of Block	Size of Array
Stage 1	$B_1 = 16$	N
Stage 2	$B_2 = 64$	N/16
Level 2	B = 1,024	N/1,024
Level 3		N/1,024*1,024

6. GPU IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we report the GPU implementation as well as evaluation of the algorithm. The experiments were run on the following systems.

—CPU: An Intel Core i7 920, with 8MB cache, 4GB RAM and a 4.8GT/s Quick path interface, with maximum memory bandwidth of 25GB/s.
—GPU: A Tesla C1060 which is one quarter of a Tesla S1070 with 4GB memory and 102GB/s memory bandwidth. It is attached to an Intel Core i7 CPU, running CUDA Toolkit/SDK version 2.2.

6.1. GPU Implementation

The size of each level is shown in Table II. In Stage 1, 16 threads are used per block of size B_1. The thread-block size is kept at 1,024 threads. Each thread finds the nearest smaller value to the left and to the right for the corresponding element. Optimality is sacrificed finding the while nearest smaller values to reduce thread divergence. Each thread searches across the 16 elements to find the nearest smaller values. Bank conflicts do not arise if half a warp (16 threads) accesses the same element of a shared memory bank. Hence, bank conflict is avoided here. The Cartesian tree is created by finding the larger valued of the two nearest smaller valued elements. For example, for a given element e, with value 5 and index i_e, the left nearest smaller value is 2 with index i_l, and the right smaller value is 3 with index i_r. The parent of i_e will be i_l, since the value at i_l is greater than the value at i_r. The left and right child of each node is also found. The tree created is traversed to give the preorder traversal of the tree. To improve parallelism, leaf nodes of the tree precompute their subtree traversal. The preorder traversal at a node if the concatenation of preorder traversal of its left and right subtrees to the node. This inherent limited parallelism is used to increase the occupancy of the threads. Arbitrarily selected nodes perform preorder traversal of their subtrees. A new tree is formed with the previously selected nodes. The sequential traversal on this tree and code generation is performed by the first thread of each block.

Our method is able to scale on the GPU, and requires limited number of registers and limited shared memory. As presented in Section 3, memory is an important problem in GPUs. Using succinct representation, we were able to reduce global memory usage. Also, the design and implementation of Stage 1 has reduced shared memory and register requirement with regard to Fischer-Heun's method. The size of the succinct representation per block for Level 1: Stage 1 is thus 32 bits. Hence, comparing elements of a block is equivalent to querying on the 32-bit word.

The size of Stage 2 is fixed at 64 elements per block so that one thread block can process both stages together. This reduces the kernel call overhead as well as global memory read/write overhead. In Stage 2, only 64 threads are active out of the 1,024 threads for each thread block. Bender's method for 64 elements requires six iterations. Thread synchronization is performed at the end of each iteration. For each element, the resulting six relative indexes can be stored by using 21 bits. To decrease space usage, results of only five iterations are used. This can be stored in 15 bits. It is notable

Table III. GPU Optimizations of Each Stage

Stage	Global Memory Usage	Thread Divergence	Shared Memory and Register
Preprocessing			
Level 1: Stage 1	Uses succinct representation	Uses nonoptimal yet symmetric methods	Low memory usage Limited bank conflicts
Level 1: Stage 2	Uses succinct representation and reuses data from Stage 1	Optimal and symmetric method	Reuse of Stage 1 memory
Level 2	Small data size	Optimal and symmetric method	Negligible bank conflicts
Querying			
Level 1: Stage 2	Maximum of 3 accesses to read succinct representation	Usage of succinct dictionary reduces divergence	Limited memory needed for succinct dictionary
Level 1: Stage 2	Up to 3 accesses of succinct representation	Symmetrical computation with low overhead	Reuses memory from earlier stages
Level 2	Reduces number of access to the data array	Limited	Reuses memory
Level 3	Metadata stored in texture memory	Limited	Reuses memory

that the querying method also requires results of five iterations. The minimum valued index for the block and its value is also stored. It should be noted that the GPU state is lost as soon as the thread block exits. Hence, before each exit, the relevant parts of the thread state has to be explicitly stored in the global memory.

Level 2 is managed as a different GPU kernel. The block size at this level is 1,024. Bender's algorithm is used without compression in this level. Bender's method is parallel in nature and allows for negligible bank conflicts. Level 3 has very limited data to process; hence, a limited number of threads are sufficient. The metadata from Level 3 is redirected to the GPU texture memory.

While querying, each query is handled by a single thread. The compressed representation at each level reduces global memory reads. In the querying phase, the number of divergent branches are limited. In all the levels, the type of queries can be (i) within the same block, (ii) in adjacent blocks, (iii) in blocks separated by a single block, and (iv) distant blocks. The number of subqueries are 1, 2, 3, and 2, respectively, for each type. In any level, the number of divergent branches are hence a maximum of 2. In Level 1: Stage 1, within a subquery, divergent branches are reduced by the usage of a succinct dictionary. *BS* is kept as 4. It is notable that the value along with the index at each of the relevant indexes are read from the global memory. Usage of succinct representation is able to reduce the number of global memory accesses. A detailed list of optimizations used to handle GPU-specific issues is presented in Table III.

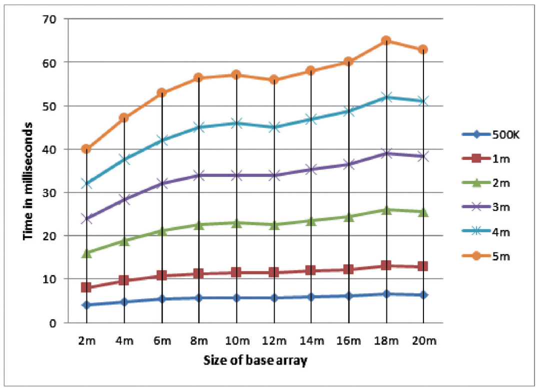
6.2. Experimental Results

Each experiment was run 10 times, and an average value is reported. Two set of results are shown, one containing the raw speed-up of the algorithm on the GPU with respect to CPU. In the second experiment, the time taken to transfer the data bidirectionally is added to the results. The first one shows the feasibility of using RMQ as a primitive for the GPU, and the second one shows the feasibility of GPU acting as an accelerator for RMQ operations for the CPU. The data is randomly generated. The obtained results are tabulated in the following text, and the number of queries is fixed at 10M. CPU times reported are for single-threaded and multithreaded (8 threads) queries. Times reported are for the 10M queries. All times are in milliseconds.

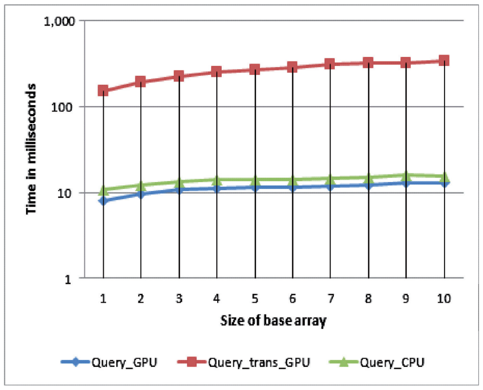
Experimental results provided are for batch mode, that is, a large number of queries are processed together. In the experimental results shown, all the queries are performed in a single CUDA kernel call on the GPU. A speed-up of 20–25 is noted against

Table IV. Results for Queries on Large Datasets

Base Array	Preprocessing time(ms)		Query Time(ms) 10M queries		
(M)	GPU	CPU	CPU-1	CPU-8	GPU
100	88	2,053	5,187	1,128	140
150	120	3,092	5,471	1,167	152
200	158	4,133	5,513	1,200	160
250	193	5,130	5,648	1,211	168



(a) GPU query time comparison over various query sizes



(b) CPU vs. GPU comparison for 1M queries

Fig. 4. Comparison of results for querying.

a sequential implementation preprocessing over preprocessing time taken by Fischer-Heun's algorithm. A speed-up of 25–35 over the implementation provided by Fischer is noted while querying. While querying, a speed-up of 7–8 over an 8-threaded version of Fischer's method is noted. For large base arrays, the time per query was nearly constant given a constant number of queries, while the size of the base array was changed per experiment. This was not true in the case of the CPU implementation, where the time taken increased over time. This can be directly attributed to the user management of the cache memory. With the addition of data transfer overhead, the speed-up with regard to an 8-threaded implementation fell to an average of 4–6. Table IV and Figure 4 show the results of our experiments. Query_GPU is the time taken by only the GPU query. Query_CPU is the comparison of Fischer-Heun's implementation provided by them. Query_trans_GPU is the time taken by our method including the CPU transfer time. As compared to the method of Fischer-Heun, our method used 7.5 bits of space per element against their 7 bits for integer data types, which use 32 bits of space for storage. The space usage includes the space required to store the minimum and the relative index from each level.

6.3. Analysis of Results

6.3.1. CPU Comparison of Results. The following results are taken on a CPU with Corei7 920, running g++ version 4.2, O3 flag has been used while compiling. The CPU version of our method is similar to the GPU implementation. Results on an array of 1 million elements is presented here. It was observed that for Fischer-Heun's method, it took 15ms for preprocessing their succinct representation. In comparison, our method using a single thread took 44ms for preprocessing. Both methods show linear scaling of time with an increase in the size of the data. A comparison of time taken for different data sizes on the CPU is shown in Table V. For Fischer-Heun's method, processing of the succinct stage takes about 70% of the total time. Our method takes about 90% of the

Table V. Comparison for Preprocessing of CPU Implementation of Both Methods

Size of Array (in million elements)	CPU-Fischer (ms)	CPU-DFUDS (ms)
1	15	44
2	31	90
10	159	454
20	328	905
50	840	2,303

time in the succinct representation. Note that this is a sequential representation of a parallel algorithm and hence is slower. The time taken for our CPU based method for the later stages are same as that of Fischer-Heun's method. This is attributed to the similarity in the methods.

In the querying phase, in Fischer-Heun's method, approximately 50% of the time was taken by the succinct representation processing.

It is the claim by Fischer-Heun that their method outperforms other succinct representation in RMQ queries. Hence, we claim to have compared our method against the current state-of-the-art method.

6.3.2. GPU Results. In the preprocessing phase, succinct processing takes 60% to 70% of the total time. The rest of the stages put together take the rest of the 30% to 40% time. The data is maintained in the GPU global memory. In the querying phase, our method takes 70% to 80% of the time in processing the succinct representation. The secondary stages take the rest of the time. It is notable that the increase in the time taken by our method with a fixed number of queries and varying size of the data is nearly constant. This is shown in Figure 4(a). As can be seen by comparing with the discussion in Section 6.3.1, our method shows similar behavior on the CPU as well as the GPU in terms of time ratios.

GPU implementation of Fischer-Heun's method is impractical due to the substantial divergence and memory requirement. A discussion on the GPU performance of Fischer-Heun's algorithm is hence not followed here.

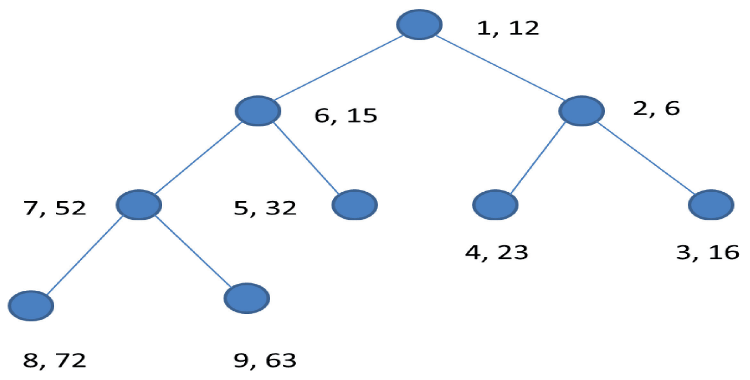
7. APPLICATIONS OF DRS

In this section, we show some applications of DRS in parallel computing. DRS on the GPU can also enable queries on array-based representations of complex structures such as trees, as shown in Section 7.1. Similarly, certain string operations such as the longest common extension can also be modeled using the DRS primitive, as shown in Section 7.2. Other applications suggested in Fischer and Heun [2006, 2007] fit on the GPU with support from our DRS implementation.

7.1. Tree Queries for the GPU as DRS

A tree can be represented by a set of arrays using the Euler tour technique [JáJá 1992]. The representation can be based on the preorder number of nodes and the left- and rightmost appearances of each node in an Euler tour of the tree. Certain queries on the tree, such as subtree queries and the least common ancestor, can be converted into queries on a range of the Euler tour. We elaborate on these two types of tree queries.

7.1.1. Subtree Queries. Finding the minimum value in a subtree can be modeled as a range query. The minimum value in the subtree of a node is equivalent to finding the minimum along the preorder traversal of its subtree. If values in a tree are arranged in an array K, according to the preorder number of each node, the minimum in the subtree of node n is the range minima in the range $[preorder(n), preorder(n) + size(n)]$.



(a) Tree with previously assigned labels and values

Label	Preorder Number	Size	Node Value	Minimum
1	1	9	12	6
6	2	5	15	15
7	3	3	52	52
8	4	1	72	72
9	5	1	63	63
5	6	1	32	32
2	7	3	6	6
4	8	1	23	23
3	9	1	16	16

(b) Corresponding table

Fig. 5. A tree and corresponding query.

Here, $preorder(n)$ is the preorder number of node n, and $size(n)$ is the size of the subtree rooted at n.

In Figure 5, a tree and all its subtree queries are shown. Each vertex in the tree has a label and a value previously assigned to it. The minimum value in every subtree can be found in $O(n)$ sequential time by performing a level order traversal. However, this method is not easily amenable to parallelism as compared to using DRS. We suggest that this method is especially useful in applications where preorder or postorder traversal has been already done and the size of the subtree is previously known. Examples of such algorithms include tree-based algorithms, such as parallel biconnected components and ear decomposition.

Results. In our implementation, the preorder and size of the subtree are assumed to be previously known. Two categories of queries are looked into:

—all nodes perform a subtree query and
—a small subset of nodes need to perform a subtree query.

These two are dealt with differently because coalescing will be much larger in the first case. This decreases the per-query time. This is especially true if the queries are arranged in the order of the preorder numbers. The comparison of perquery time is given in the table for different number of queries on a tree size of 20M. The queries are randomly chosen, and unsorted. For comparison with sequential implementation, a sequential expression evaluation algorithm, which performs a comparison of the key

Table VI. Comparison of Complete Queries

Tree	Query Time (ms)	
Size (M)	GPU	CPU
1	6.84	48
2	16	99
4	37	203
8	90	427
16	194	817

Table VII. Comparison of Different Numbers of Queries on a Tree with n Nodes

Size	GPU Query Time (ms)
100*log(n)	0.1
n/100	2
n/10	15

Table VIII. Results of LCA on Preprocessed Trees

Tree Size (M)	10	11	12	13	14	15
Time Taken (ms)	6.8	7.3	8.0	8.7	9.2	10.1

at the node to the minimum of subtree is done. The time taken for a tree of 20 M nodes is 1s.

For complete queries where the number of queries is equal to the number of nodes, Tables VI and VII show the query time for both cases.

7.1.2. Least Common Ancestor as DRS. The least common ancestor (LCA) has a direct dual with DRS. Applications that map LCA to DRS have been shown in Fischer and Heun [2007]. As mentioned in Section 7.1, the Euler tour technique is used as a preprocessor for trees, helping to form efficient algorithms, especially in the context of the GPU. The level order and preorder traversal are computed using the Euler tour technique. Euler tour along with list ranking of Rehman et al. [2009] and the scan primitive [Sengupta et al. 2007] are used to preprocess a given tree to generate the required traversal of the graph [JáJá 1992]. Given an Euler tour of a tree T, let the rank of an edge e be defined as the position of e in the Euler tour of T. For a node i, let $L[i]]$ and $R[i]$ denote the left- and rightmost occurrences of node i in the Euler tour of T. Suppose the edge uv has a rank of i. We define an array A as $A[i] :=$ level(u), where $level(u)$ denotes the level to which node u belongs in the tree. Then, it holds that the least common ancestor of two nodes u and v can be computed as follows. Let $k = RMQ_A(R[u], L[v]$. Then, LCA(u, v) $= w$ such that the edge wx has a rank of k.

A detailed commentary on ETT for the GPU is given in Rehman [2010] with a method to generate ETT, R, L and A for a given tree. The array A is to be processed for range queries.

Results. The times (independent of the ETT step) found for a $\frac{n}{10}$ random queries on a tree of size n (in millions) are given in Table 7.1.2.

7.2. Longest Common Extension for the GPU

Longest common extension (LCE) two strings is their longest common prefix. Though trivial using suffix trees, it can also be found by querying on a suffix array. A suffix array representation of a string typically consists of indices corresponding to lexicographically sorted suffixes, stored in an suffix array (SA). For example, for a string "SAAD," the corresponding values in the suffix array is 2(AAD), 3(AD), 4(D), 1(SAAD). Additional supporting data structures assist computation. Inverse suffix array, SA^{-1}, contains the lexicographical rank of each element in the string. For example, for "SAAD," inverse suffix array will contain 4(SAAD), 1(AAD), 2(AD), 3(D). The longest common prefix

(LCP) contains the length of the maximum common prefix of adjacent elements in the suffix array.

String:	S	A	A	D
SA	2	3	4	1
SA^{-1}	4	1	2	3
LCP	0	1	0	0

The LCE of substrings starting from indices i, j is given by the following equation: $LCE[i, j] = LCP[RMQ_{LCP}(SA^{-1}[i] + 1, SA^{-1}[j])]$

For example, the LCE of string AAD and AD is equal to $RMQ[2, 2]$=1. Assuming an application that requires large number of such queries on a static string, our method can be used to preprocess the LCP array, so that LCE queries can be answered efficiently.

Results. The suffix array and the LCP array are generated on the CPU and the LCP array is moved to the GPU. LCP array acts as the base array for Range queries. The query time for LCE computation on a string of size 1M with 100K queries is 0.52ms.

8. CONCLUSION AND FUTURE WORK

We have presented here, an implementation of a succinct representation on the GPU which uses $2n$ bits of space for a tree of size n. This representation can represent Cartesian trees. Additionally, the succinct representation is useful in answering LCA queries on Cartesian trees. The additional space required is constant and is sufficient to answer LCA queries in constant time. The succinct data structure is amenable to the GPU model of computation. This method can naturally scale to a multi-GPU environment, such as a cluster of CPUs with GPUs attached to each CPU.

Range minima queries are implemented using the succinct representation. The method we designed for range minima querying has low space overhead and is able to provide a speed-up of more than 25 over a CPU for batch querying. The range minima algorithm we presented can be used for applications such as tree queries and string queries. We also recommend that DRS can be helpful in implementing tree based graph algorithms on the GPU.

In the future, we plan to explore succinct representations that are amenable to the GPU as well as reduce the number of bits required.

REFERENCES

Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., et al. 2006. The landscape of parallel computing research: a view from berkeley. Tech. rep., UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bayoumi, A., Chu, M., Hanafy, Y., Harrell, P., and Refai-Ahmed, G. 2009. Scientific and engineering computing using ati stream technology. *Comput. Sci. Eng. 11,* 6, 92–97.

Bender, M. A., Pemmasani, G., Skiena, S., and Sumazin, P. 2001. Finding least common ancestors in directed acyclic graphs. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*. SIAM, Philadelphia, 845–854.

Bentley, J. and Friedman, J. 1979. Data structures for range searching. *ACM Comput. Surv. 11,* 4, 397–409.

Berkman, O. and Vishkin, U. 1989. Recursive*-tree parallel data-structure. In *Proceedings of the 30th IEEE Annual Symposium on Foundation of Computer Science*. 196–202.

Cederman, D. and Tsigas, P. 2008. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th Annual European Symposium on Algorithm (ESA'08)*. Lecture Notes in Computer Science, vol. 5193, Springer, Berlin, 246–258.

Fischer, J. and Heun, V. 2006. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combination Pattern Matching (CPM'06)*. Lecture Notes in Computer Science, vol. 4009, Springer, Berlin, 36.

Fischer, J. and Heun, V. 2007. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proceedings of the 1st International Symposium on Combinatiorics,* *Algorithms, Probablistic and Experimental Methodologies (ESCAPE'07)*, Lecture Notes in Computer Science, vol. 4614, Springer, Berlin, 459–470.

FISCHER, J., HEUN, V., AND STIIHLER, H. 2008. Practical entropy-bounded schemes for o (1)-range minimum queries. In *Proceedings of the Data Compression Conference (DCC'08)*. IEEE, Los Alamitos, CA, 272–281.

GABOW, H., BENTLEY, J., AND TARJAN, R. 1984. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*. ACM, New York, 135–143.

JÁJÁ, J. 1992. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA.

JANSSON, J., SADAKANE, K., AND SUNG, W.-K. 2007. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA'07)*, ACM, New York, 575–584.

NVIDIA, C. 2007. Compute Unified Device Architecture Programming Guide. NVIDIA, Santa Clara, CA.

PATIDAR, S. AND NARAYANAN, P. 2009. Scalable split and gather primitives for the gpu. Tech. rep., IIIT/TR/2009/99.

REHMAN, M. 2010. Exploring irregular memory access applications on the GPU www.contrib.andrew.cmu.edu/~suhailr/thesis.pdf.

REHMAN, M., KOTHAPALLI, K., AND NARAYANAN, P. 2009. Fast and scalable list ranking on the GPU. In *Proceedings of the 23rd International Conference on Conference on Supercomputing*. ACM, New York, 235–243.

SADAKANE, K. 2007. Compressed suffix trees with full functionality. *Theory Comput. Syst. 41,* 4, 589–607.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE, Los Alamitos, CA, 1–10.

SCHEUERMANN, T. AND HENSLEY, J. 2007. Efficient histogram generation using scattering on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM, 33–37.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. 2007. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Eurographics Association, 106.

SINTORN, E. AND ASSARSSON, U. 2008. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput. 68,* 10, 1381–1388.

SOMAN, J., KUMAR, M., KOTHAPALLI, K., AND NARAYANAN, P. 2010. Efficient discrete range searching primitives on the gpu with applications. In *Proceedings of the International Conference on High Performance Computing (HiPC'10)*. IEEE, Los Alamitos, CA, 1–10.

STONE, J., GOHARA, D., AND SHI, G. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng. 12*, 66.

Received August 2011; revised June 2012; accepted July 2012