Efficient Discrete Range Searching Primitives on the GPU with applications

Jyothish Soman, Matam Kiran Kumar, Kishore Kothapalli and P J Narayanan
International Institute of Information Technology
Gachibowli, Hyderabad, India-500032
{jyothish@students.,kiranm@research.,kkishore@, pjn@}iiit.ac.in

Abstract

Graphics processing units provide a large computational power at a very low price which position them as an ubiquitous accelerator. Efficient primitives that can expand the range of operations performed on the GPU are thus important. Discrete Range Searching(DRS) is one such primitive with direct applications to string processing, document and text retrieval systems, and least common ancestor queries.

In this work, we present a GPU specific implementation of DRS with an optimal space-time trade off. Toward this end, we also present GPU amenable succinct representations and discuss limitations on the GPU. Our method uses 7.5 bits of additional space per element. The speedup 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 methods obtain a speedup of 6-8.

We study applications of the DRS on the GPU. Also, we suggest that most graph algorithms which focus on using least common ancestor, can easily be enabled on the GPU based on range minima primitive. Beyond this, we show applications of DRS in string querying and tree queries, and suggest how DRS can be helpful in implementing tree based graph algorithms on the GPU.

1. Introduction

The programmability and prevalence of the GPUs has opened up avenues for using the GPU for applications beyond the data parallel comfort zone of the GPU. The competitive cost to performance ratio has expanded the usage of the GPU into general purpose computing. General purpose computation on the GPU (GPGPU) has positioned the GPU as a coprocessor for general purpose acceleration. A large body of data parallel computations can be accelerated using the GPU. Programming models for the GPU include CUDA, STREAM and OpenCL. The programming models are able to make good use of the computational resources of the GPU. The performance is mainly because these programming models are tightly bound to the architecture of the GPU.

The GPU can be considered as a collection of groups of weak processing elements. Each group has a limited amount of shared memory, and each group is connected to the global memory. This global memory also acts as an interface between the CPU and the GPU. A more detailed description of the GPU architecture follows in section 2. GPUs are suited for fine grained data parallel applications. Thus the application design for a GPU is different from that of the CPU. 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. GPU applications heavily rely on these data primitives, which perform data parallel tasks efficiently. Common primitives include `scan`[15], `sort`, `split` and `list ranking` [14]. Due to the unique architecture of the GPU, architecture specific primitives are required. A large body of work for the GPU currently is focused on forming efficient primitives.

Hence, GPU programming is thus a primitives bound processing mechanism. One such primitive is the Discrete Range Searching (DRS), which has wide spread applications in graph theory, string processing, VLSI, document retrieval and biology. Though the large body of applications are largely due to the dual between least common ancestor and range minima, discrete range search can be considered an application in itself, especially in the context of the GPU. For example, in Section 6.1, we present ways in which the tree queries on the GPU can be answered efficiently using DRS.

Discrete range search in the current context encompasses two operations, range minima query and range maxima query. It should be noted that the two are fundamentally the same operations, with a difference in operator. Among a variant of these primitives is maximum sum subsegment query which is an application of the range minima query and range maxima query, the mapping is shown in [7]. Given an array A, $|A| = n$, the formal definition for the two are as given below:

`Range minima query`:
$RMinQ_A[i,j] = k, : A[k] \leq A[l],\ k,l \in [i,j], 0 \leq i,j < n$

`Range maxima query`:
$RMaxQ_A[i,j] = k : A[k] \geq A[l],\ k,l \in [i,j], 0 \leq i,j < n$

There are many variants to the above. The one dis-

978-1-4244-8520-8/10/$26.00 ©2010 IEEE

cussed here performs offline querying on a static array. A generic approach followed by all algorithms belonging to this category is to preprocess the base array to precompute answers to queries. One method is to explicitly precompute all possible queries. This process can be made more efficient by dividing a query into a group of smaller queries and preprocessing the answers to each smaller query.

1.1. Related Work

Range minima is an important primitive especially because of its dual with least common ancestor(LCA) [6], [4], [10]. Many solutions exist for the problem. A parallel algorithm for LCA was presented by Alstrup *e*t al [2], which focussed on solving the LCA problem in parallel. Their solution creates a unique label of length $\Theta(\log n)$ for each node such that the label of the least common ancestor of nodes i, j can be found by comparing their labels. This method in the RMQ context requires processing the Cartesian tree of a given array. The labels L of each element and the corresponding inverse L^{-1} needs to be stored, here $L^{-1}L[i] = i$. The total number of bits required for RMQ based on this method is thus greater than $2n\log(n)$. Another method is presented by Gabow[10], their work presents a strong dual between LCA and RMQ was presented, and suggested that solving one would generate a solution for the other. This was the basis of the algorithm by Berkman-Vishkin [6]. A parallel algorithm for this problem was first suggested by [6]. Berkman's method is a multilevel algorithm. For an array A of size S, at every level i, the array A is divided into blocks of size 2^i, $0 < i < \log(S)$. The suffix and prefix minima of each block of the array is stored. While querying, let i, j $(i < j)$ be a query, a suitable level k is found such that $(i \bmod 2^k) + 1 = j \bmod 2^k$. The minimum of the prefix minima j and suffix minima of i in level k is the solution to the query. A simpler and more practical version of the same approach is presented by Bender *e*t al [4]. Bender *e*t al presented 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 is preprocessed to form a lookup table M, such that M[i][j] 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 by finding the minimum element in two overlapping blocks,$[i : i + 2^k - 1][j - 2^k + 1 : j]$.

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

To reduce the space requirements, Fischer *e*t al [8], [9] presented a succinct representation based solution that reduces space requirement by representing a tree using its succinct representation. The technique presented in [8] has comparable per query time with other methods in literature. The time per query for [9] was shown to be marginally higher than the method by Alstrup *e*t. al.[2]. The space requirement is much smaller for the former. Accelerated cascading was used for further reducing the space requirement. A detailed discussion follows in Section 3.

The rest of the paper is arranged as follows, section 2 presents the GPU computation model, section 3 presents a discussion on Fischer-Heun's algorithm. Section 4 presents our method. Section 5 presents our results, section 6 presents some applications for the GPU and the corresponding results.

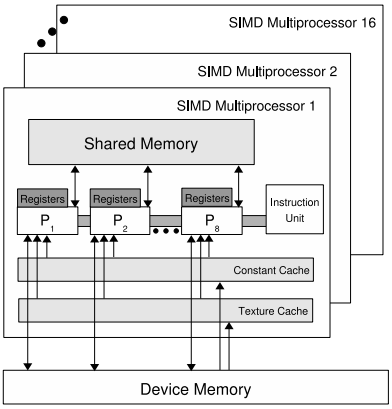
2. GPU Computation Model

The GPU is a massively multi-threaded architecture containing hundreds of processing elements (cores). Each core comes with a 4 stage pipeline. 8 Cores are grouped in SIMD fashion into a Symmetric Multiprocessor (SM), hence each core in an SM executes the same instruction. Each Quarter of a Tesla S1070 is a Tesla C1060. Tesla C1060 has 30 SMs, and a total of 240 processing cores per Tesla C1060.

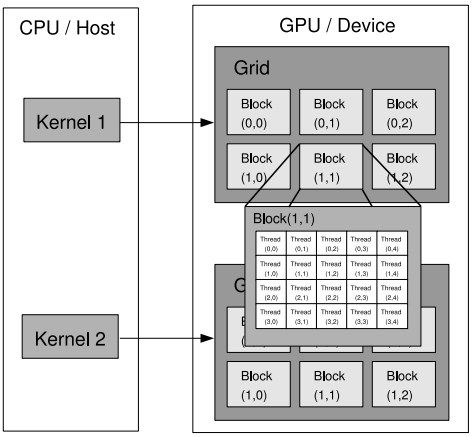
The CUDA API allows a user to create large number of threads to execute code on the GPU. Threads are also grouped into blocks and blocks make up a grid. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called warps, each containing 32 threads. An SM executes 1 warp at a time. CUDA has zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp. For this purpose, NVIDIA recommends at least 512 threads be assigned to an SM to keep an SM fully 'occupied'.

The GPU also has various memory types at each level. A set of 32-bit Registers is evenly divided among the threads in each SM. 16 Kilobytes of Shared Memory per SM acts as a user-managed cache and is available for all the threads in a Block. The Tesla C1060 also comes with 4 GB of off-chip Global Memory which can be accessed by all the threads in the Grid, but incurs about hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the constant memory and texture memory for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit kernels. Prior to launching the kernel, all the data required for the computation must be transferred from the Host (CPU) memory to the GPU (Global) memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronisation for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronisation of all threads can only be performed across separate kernel launches.



(a) The CUDA GPU Memory Hierarchy



(b) The CUDA Computation Model

Figure 1. CUDA

Resource or Configuration Parameter	Limit
No of Cores	240
Cores per SM	8
Threads per SM	1,024 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	16,384 registers
Active Warps per SM	32 warps

Table 1. Constraints of Tesla C1060 on CUDA

3. Fischer-Heun's algorithm

Fischer-Heun's Algorithm [8], [9] is a hierarchical solution for DRS with $\Theta(n)$ space and $\Theta(1)$ time for query. A given query is divided into smaller queries and the partial results are combined together to find the complete result. For example, let the method have three levels, 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

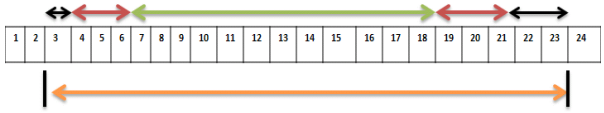


Figure 2. Decomposition of a query, here b2=6,b1=3, query=3,23

An example of the same is shown in Figure 2. Here b1=3,b2=2. The resulting queries are $\langle 3,3 \rangle, \langle 4,6 \rangle, \langle 6,18 \rangle, \langle 18,21 \rangle and \langle 22,23 \rangle$.

As seen in the description above, the query q_1 and q_5 lie in blocks of size b_1. In the algorithm of Fischer-Heun, b_1=log$(n)/4$ and each such block is represented by a Catalan rank. 1<Rank(block)$< C(k)$, where $C(k)$, is the k^{th} Catalan number, k=log$(n)/4$. For each Catalan number, all possible DRS queries are preprocessed and stored so that any DRS query can be answered in $\Theta(1)$ time. The resulting lookup table hence is of size ${}^kC_2 \cdot C(k)$.

The next array is formed from the minimum element of each block. This array is then broken down into blocks of size of b_2. Each block of size b_2 is preprocessed by the method suggested by Bender *et. al.* [4]. The minimum element of each block is taken. For the new array, the same process is repeated. An example of such a decomposition is shown in Figure 3.

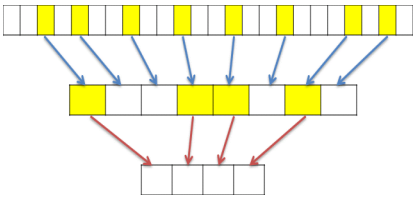


Figure 3. Reduction in the size of array based on b1,b2, here b1=3,b2=2. yellow coloured element is the minimum in a block

Space taken in Fischer-Heun's solution for the lookup array increases with the size of the base array. Hence a small block size is selected so that the lookup array is small. The small size of the block which they have enforced in their method does not reduce the problem size appreciably. This causes an increase in the number of elements in the upper levels, and hence scalability becomes an issue.

4. Discrete Range Search for The GPU

In Fischer-Heun's algorithm, as presented in Section 3, each query is independent, hence for a large number of queries there is inherent data parallelism in the querying phase. In the preprocessing stage each block can be processed in parallel. Also, intra block parallelism can be found. For example, Cartesian tree generation can be done in parallel using the all nearest smaller value algorithm of Vishkin[5].

In practice, parallel algorithms are specific to the underlying hardware. The efficient utilisation of available resources on the platform is a major factor in the applicability of a given algorithm. Also other factors such as space utilisation need to be addressed. This section describes our algorithm for DRS, with focus on Range Minima Query (RMQ) on the GPU. The algorithm of Fischer-Heun [9] is modified in our method to fit the computational model of the GPU. The major factors that have been taken care of in implementing the algorithm on the GPU are the following:

- Adapting to the computational model of the GPU
- Iteratively reducing the search space
- Reducing random reads
- Presenting data structures with succinct representations, thus reducing number of reads for processing of queries, as well as reducing space requirement.
- Maximising shared memory usage on the GPU.

The algorithm presented here, divides a query into multiple smaller queries. In the context of the GPU, this method assists shared memory based processing and enables GPU centric optimizations. Each level reduces the size of the base array, by dividing the array into blocks of fixed size. This method is known as accelerated cascading. Repeated use of accelerated cascading hierarchically reduces the problem size and space utilisation. In our solution, at each level, the size of the base array is reduced by a factor of B. 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 that the first level requires the maximum space. In the first level, the first stage uses a space optimal succinct representation to reduce space requirements. The second stage uses a weaker but faster succinct representation. The third level utilizes the small size of array to utilize hardware cache available on the GPU.

4.1. Succinct representation for the GPU

Succinct representations helps reduce space requirements, while allowing multiple operations and queries. Space requirement is high in the early stages of the algorithm. In the context of the GPU, this is important, as the RAM/Global memory available on a single GPU is limited, hence space saving solutions are important for scalability of the result. The relative ordering of elements in an array can be represented by a Cartesian tree. The Cartesian tree can be stored succinctly, thus space requirements can be reduced. This approach has also been suggested by Fischer-Heun[8]. Succinct representations on a CPU is straightforward, but on the GPU, some architecture specific bottlenecks arise, some of them are as follows:

1) The number of registers available on the GPU per thread is limited. 16384 registers are available per SM on a Tesla C1060. If the total register requirement per thread block exceeds 16384, then the register allotment overflows into much slower memory. Reduction in the number of threads per thread block, to increase the register count per thread, leads to decrease in the number of concurrently active threads per SM. Both can cause performance loss.
2) Support for datatypes with less than 32 bits of size on the GPU is limited. The hardware is more focused on floating point and integer datatypes. Operations on characters are treated as integer operations, hence no performance improvements arise in operating using characters on the GPU.
3) Thread divergence is another factor that decides the performance on the GPU. Algorithms that cause large number of divergent branches to be present per warp cause the performance to degrade.
4) Lack of a large cache per processor imposes performance penalty when large static data is used by each thread. Hence lookup table based methods have to be carefully designed.

We present some first results on succinct representation of Cartesian trees for the GPU.

4.2. Succinct Representation for Level 1

Level 1 is divided into two stages. In Stage 1, array A is divided into blocks of size $B1$. Each block is then represented by a Cartesian tree. The Cartesian tree is stored succinctly. The minimum of each block is stored in array A'. In Stage 2, array A' is divided into blocks of size $B2$. Array A' is preprocessed to answer each query.

The representation of an array as a Cartesian tree, and the storage of the Cartesian tree succinctly is done by using a Preorder traversal based balanced parenthesis representation. The same is explained here.

4.2.1. Succinct Representations for Stage 1: Preorder Balanced Parenthesis. In succinct representations presented in [11], each node is represented by a one, and an empty node is added for every absent child node. Hence each node of the original tree now has two children. Such a tree can be referred to as a padded tree. An example is shown in Figure 4. We note that performing a preorder traversal on the padded tree will produce a representation more suitable to the GPU. The resulting string has $2n+1$ bits of space. Neglecting the trailing zero, the representation uses exactly

$2n$ bits of space. Balanced parenthesis as presented in [3] cannot represent Cartesian trees. We have improved upon their representation, our method can support Cartesian trees. In our method, a parenthesis covers only the left subtree of a node instead of the whole subtree of a node. Thus nodes inside a parenthesis are the left children. Nodes falling inside the parenthesis of the parent, but not inside a nodes parenthesis, form its right subtree. Such a method guarantees Cartesian tree support.

Given an array A, its Cartesian tree can be found by finding for each element, the index of nearest node on the left and right with value smaller than its own. between these two, the parent of the element has the larger stored value in the array. The succinct representation can be formed by combining mutually independent partial traversals of the tree. Each node with 2 children is assigned as a splitter. The tree is then split at these splitters. It can be noted that at each splitter node s, the succinct code at s $SC(n)$ can be given by the concatenation of 1, $SC(lt)$, $SC(rt)$, where lt and rt are the left and right child nodes of s. We summarise the succinct representation generation algorithm in Algorithm 1.

Preorder Balanced Parenthesis can generate RMQ information by linearly parsing through the representation once. As the tree is Cartesian, range minima of two nodes in a given array is equivalent to their least common ancestor in the Cartesian tree. In our representation, the n^{th} node in the array will be represented by the n^{th} zero in the bit sequence. Least common ancestor, lca(i,j), of i, j is the node with i in its left subtree, and j in the right subtree. Let I be the position of the i^{th} zero and J be the position of the j^{th} zero. The lca(i,j)=k is thus the node with i in its left subtree, and j in its parent's left subtree. At the k^{th} zero, the left subtree of i closes. We further state that at bit position K=bit position of k^{th} zero ,$I <= K <= J$, will have the minimum difference between the number of ones and zeros to its left in the bit string. The left most such bit position is taken in case of multiple results with equal minimum value. The array index is the number of zeros to the left of the bit position.

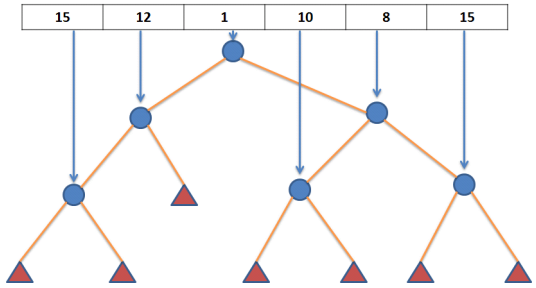
To support RMQ queries on the bit string, we propose that instead of adding additional data to each string, we can preprocess all bit sequences of length BS in the query phase and store in the caches. This produces a strong upper bound of 2 bits of global memory usage per node, while supporting constant time querying. For bit strings of size BS, we store for all possible bit strings, the number of zeros in the string, the minimum difference between the number of ones and zeros in the string. We do the same for strings of size $BS/2$. It should be noted that BS can be any divisor of n such that $BS/2$ is also a divisor of n. Such an arrangement saves us from traversing the bit string and reducing divergence.

The algorithm is stated in Algorithm 3.

4.2.2. Succinct representations for the 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 outweighs the space advantage. Hence alternative methods, which are space intensive, but can answer queries fast are used. The method suggested by Bender [4] is used here. The Block size is taken as B_2. Iteratively, 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 $i+k$ was found, and the relative index w.r.t. i is stored. For blocks of size B_2, $\frac{\log(B_2)\cdot\log(B_2-1)}{2}$ bits are required for storing the result. For an element 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 $i+2^k-1$. Minimum in the range i to $2^{k-1}-1$ and $i+2^{k-1}$ to $i+2^k-1$ has already been computed previously, the minimum in the range i to 2^k-1 can be directly computed. Let j be the minimum in the range i to 2^k-1, the relative index of j w.r.t. i need to be stored. For each element, the resultant relative indexes are packed into a single word. Compared to explicitly storing the indexes and minimum values, this method is used to save space.

Algorithm 1 Preprocessing For Level1:stage1

For Block b
for Each e ∈ b, Find Nearest smaller values and form tree
for Each node e, if deg[e]=3, e ∈ splitter
Include root as splitter
Split tree such that each splitter is a root
for Each splitter
Perform preorder traversal to form partial bit sequence
for Each splitter e, form new tree
Parent[e]=lowest ancestor la s.t. la∈ splitter
Perform sequential traversal to form full preorder bit sequence



Code:1110001100100

Figure 4. Conversion of an array into its succinct representation using an intermediate tree representation. Here blue circles represents actual tree nodes, red triangles represents dummy nodes

Algorithm 2 Preprocessing algorithm for DRS

Level 1:Stage 1
for Each block k of size B_1
  Perform Algorithm 1 on block k and store bit sequence in bit_sequence[k]
  Pass minimum value A'[k] to next level
Level 1:Stage 2
for Each block k of size B_2
  val[e,0]=A'[k·B_2+e],0≤ e≤B_2-1
  for i=1:log(B_2)
    for Each element e
      val[e,i]=min(val[e,i-1],val[e+2^i-1,i-1])
      Concatenate index[val[e,i]] to $S_1[e]$
  C2[k,e]=$S_1[e]$
Remove minimum value array from previous level
Pass minimum value L2[k] to next level
Level 2
for Each block of size B
  for i=1:log(B)
    for Each element e
      Find minimum each mini block $e, e+2^i$
      Store index of minimum in $C3[e,i]$
Level 3
Number of blocks =1
Block Size $B_4 = \frac{N}{B_1 \cdot B_2 \cdot B}$
Perform operation similar to Level 1:stage 2
Store result in C4

4.3. Preprocessing for Level 2 and Level 3

The method followed in the previous two stages was an index centric method. This was due to the space constraints. For each sub query, the index and the corresponding data has to be read. The number of reads from global memory required is thus 4 for each sub query in Level 1 stages. We reduce that to 3 by following a data centric method. Instead of finding indexes and then comparing the data, the data is compared first, and then the index of the minimum is found. In Level 2, the data size has reduced by a factor of B. 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. In Level 2, the algorithm of Bender [4] is used, without any succinct representation. Here the index of the minimum of each block and the corresponding index is stored. The relative index to the beginning of the block is stored. Hence $\Theta(\log^2(B))$ space is used per element for storing the indexes.

Level 3 has a small number of elements. Hence using the texture memory as a hardware cached memory can accelerate querying. Hence compact representations are crucial to maximize the cache based performance. The representation presented in stage 2 of level 1 can be reused here. It should be noted that for all the other stages, we use the global memory to maintain preprocessed data for the query phase.

The brief description of the overall algorithm is presented in Algorithm 2.

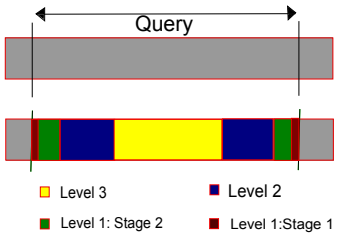


Figure 5. The decomposition of a query into multiple levels

4.4. Querying on the GPU

Querying on the GPU is trivially based on the preprocessing algorithm as presented in Figure 2. A sample query is shown in the figure 4.3. It can be noted the nature of the querying is very similar to that of Fischer-Heun's method [8], [9]. A framework algorithm is presented in Algorithm 6, the RMQ query in level1:stage1 is shown in 3. The RMQ query in level1:stage2 and level 3 form a similar pattern. RMQ query for level1:stage2 is given in 4. The RMQ query for level 2 is given in the algorithm 5.

Algorithm 3 Algorithm To Find RMQ in Level1:stage1 (i,j),i<j

FindRMQ(i,j,bit_sequence)
Index=0,Count=0,bit_index=0
Divide bit sequence into blocks of size BS
Find Block[i] containing i^{th} zero and find Block[j]
Find bit position of i^{th} zero (Bp[i]) in Block[i], and Bp[j]
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 right of Bp[j]+1 in Block[j]
Return zerocount to left of mp+1

5. Experimental Results

In this section, we report our evaluation of the base algorithm. The experiments were run on the following systems:

- **CPU:** An Intel Core i7 920, with 8 MB cache, 4 GB RAM and a 4.8 GT/s Quick path interface, with maximum memory bandwidth of 25 GB/s.
- **GPU:** A Tesla C1060 which is one quarter of a Tesla S1070 with 4 GB memory and 102 GB/s memory

Algorithm 4 Level1:Stage2 RMQ query, FindRMQ_stage2(i,j)

```
block1=i/B2,block2=j/B2
if block1=block2 then
   v1=FindRMQ_2(i mod Bk, j mod Bk)
   Return v1
end if
v1=FindRMQ_2(i mod Bk , Bk-1)
v2=FindRMQ_2(0 , j mod Bk)
Return(minarg(Arr[v1],Arr[v2]))
```

FindRMQ_2(i,j)

```
k:2^k+i<j<2^(k+1)+i
min1=Arr[min[i][k]], min2=Arr[min[j-2^k][k]]
if min1<min2 then return min[i][k]
else return min[j-2^k][k]
```

Algorithm 5 Level2 RMQ query, FindRMQ_level2(i,j)

```
block1=i/B3,block2=j/B3
if block1=block2 then
   v1=FindRMQ_3(i mod Bk, j mod Bk)
   Return v1
end if
v1=FindRMQ_3(i mod Bk , Bk-1)
v2=FindRMQ_3(0 , j mod Bk)
Return(minarg(Arr[v1],Arr[v2]))
```

FindRMQ_3(i,j)

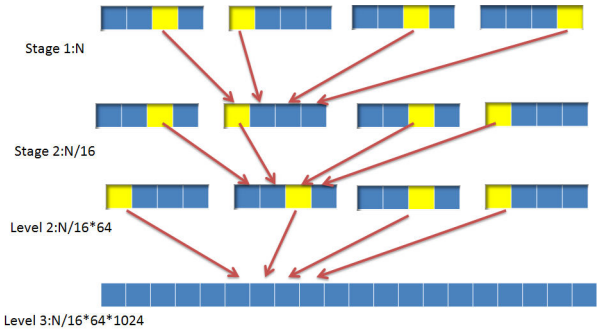
```
k:2^k+i<j<2^(k+1)+i
min1=Arr[i+code[i,k]], min2=Arr[i+code[j-2^k,k]]
if min1<min2 then return i+code[i,k]
else return i+code[j-2^k,k]
```

Algorithm 6 Algorithm For RMQ(i,j),i<j

```
block1=i/B1,block2=j/B1
if block1=block2 then
   v1=FindRMQ(i mod B1, j mod B1,
   bit_sequence[block1])
   RMQ = v1
end if
v1=FindRMQ(i mod B1 , B1-1, bit_sequence[block1])
v2=FindRMQ(0 , j mod B1, bit_sequence[block2])
i=(i/B1)+1;j=(i/B1)-1
if i>j then RMQ=ARG_MIN(v1,v2)
v3=FindRMQ_stage2(i,j)
i=(i/B2)+1;j=(i/B2)-1
if i>j then RMQ=ARG_MIN(v1,v2,v3)
v4=RMQ_level2(i,j)
i=(i/B3)+1;j=(i/B3)-1
if i>j then RMQ=ARG_MIN(v1,v2,v3,v4)
v5=RMQ_level3(i,j)
RMQ=ARG_MIN(v1,v2,v3,v4,v5)
```

bandwidth. It is attached to a Intel Core i7 CPU, running CUDA Toolkit/SDK version 2.2.

While implementing the range minima query, for each of the levels, the parameters are shown in figure 6



(a) Size reduction of base array over levels, yellow represents minimum element of a block, size of each level is given adjacent to array at each level, Block size is symbolically represented here by 4, real values are given below

Stage		Size of block	Size of Array
Stage 1	B_1	16	N
Stage 2	B_2	64	N/16
Level 2	B	1024	N/1024
Level 3			N/1024*1024

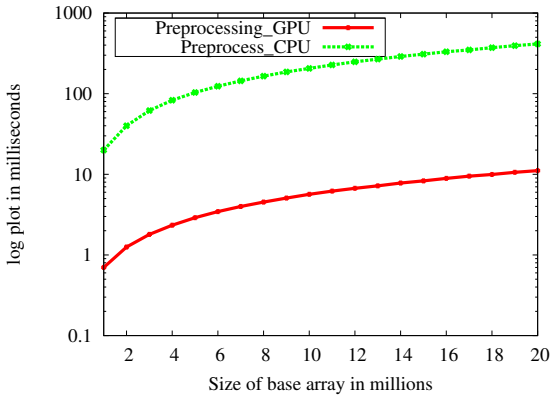
(b) Size of each stage/level and number of elements processed at each level/stage

Figure 6. Parameters for RMQ

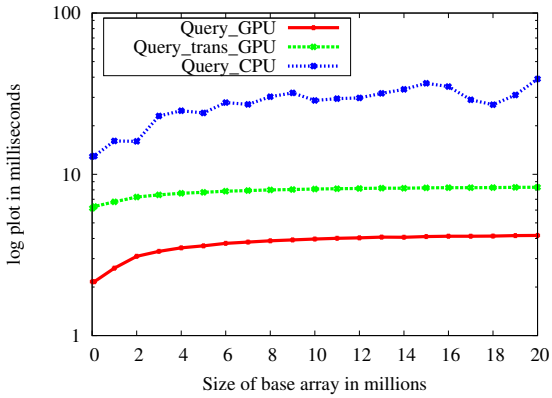
Preprocessing: The size of the succinct representation per block for level 1:stage 2 is 32 bits. Hence comparing elements of a block is equivalent to querying on the 32 bit word. The size of Stage 2 is fixed at 64 so that one thread block can process both stages together. This reduces I/O requirement. 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 different GPU kernel. The metadata from Level 3 is redirected to the hardware cache memory.

The algorithm is implemented using CUDA for the GPU. As compared to the method of Fischer-Heun, our method used 7.5 bits of space per element against their 7 bits. A CPU implementation of Alstroup's method uses approximately 7 times more space. A GPU implementation of Alstroup's method is non trivial, hence a discussion on the same is not taken up here.

Each experiment was run multiple times, and an average value is reported. Two set of results are shown, one containing the raw speedup of the algorithm on the GPU w.r.t. 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



(a) Preprocessing time for arrays of different sizes



(b) Query time for 1M queries on base array of varying sizes

Figure 7. Results for preprocessing and querying

of GPU acting as an accelerator for RMQ operations for the CPU. The obtained results are tabulated below, the number of queries is fixed at 10M. CPU times reported are for single threaded and multithreaded (8 threads). Time reported are for the 10M queries. All times are in milliseconds.

Base Array	Preprocessing time(ms)		Query Time(ms)		
(M)	GPU	CPU	CPU-1	CPU-8	GPU
100	88	2053	5187	1128	140
150	120	3092	5471	1167	152
200	158	4133	5513	1200	160
250	193	5130	5648	1211	168

To provide a just comparison, experimental results provided are for batch mode. A speedup of 20-25 is noted against sequential for preprocessing over preprocessing time taken by Fischer Heun's algorithm. A speedup of 25-35 over the implementation provided by Fischer is noted while querying. While querying, a speedup of 6-8 over an 8 threaded version of Fischer's method is noted. For large base arrays the time per query were 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 attribute to the user management of the cache memory. With the addition of data transfer overhead, the speedup w.r.t. an 8 threaded implementation fell to an average of 4-6. Figure 7 shows the results of our experiments, Query_GPU is the time taken by only the GPU query, Query_trans_GPU includes transfer times. Query_CPU is the comparison of Fischer-Heun's implementation provided by them. A detailed comparison of other available methods can be done by comparing the per query time provided in [8].

6. General applications of DRS and implications on the GPU

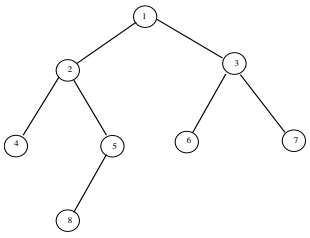
Applications of discrete range searching on the GPU have a larger scope compared to the applications on the CPU. DRS on the GPU can also incorporate queries on array based representation of complex structures. Application areas such as trees are especially important. A tree can be represented meaningfully in a set of arrays using the Euler tour technique. Hence being represented by its preorder number and leftmost and rightmost positions in the Euler tour. This causes the hierarchical structure present in the tree to be represented by a set of lists. A more detailed discussion on tree queries on the GPU follows in the Section 6.1. Thus queries on the tree can be converted into queries on a range of the Euler tour. Other applications suggested in [8], [9] fit on the GPU with support from our DRS implementation.

6.1. Tree Queries for the GPU as DRS

Given a labelled tree, such that each node has a corresponding key, in an auxiliary array K, containing the keys arranged according to the preorder number $P[i]$, i∈V, of the nodes. Tree queries such as minimum key in the subtree of a node of a tree, is equivalent to finding the minimum in an array K, in the range $P[i], P[i] + SubtreeSize(i)$. Hence,

$Subtree_{min}(i) = RMQ_K(P[i], P[i] + SubtreeSize(i))$

In the Figure 8, a tree and all its subtree queries are shown. Here, i represents the label of each node. PN represents the preorder number. S represents the size of the subtree rooted at a given node. V is the key stored in each node, and M represents the minimum key in the subtree of each node. It can be seen that the property that we suggested easily holds for the tree. Though the number of computational operations increase, as compared to a leaf to root level order traversal, the number of synchronisations among threads reduce appreciably. Such a property makes it suitable for



(a) Base Tree

i	PN	S	V	M
1	1	8	6	1
2	2	4	7	1
6	3	1	8	8
7	4	2	1	1
8	5	1	2	2
3	6	2	3	3
4	7	1	4	4
5	8	1	1	1

(b) Corresponding table

Figure 8. A tree and corresponding Query

massively multithreaded architectures such as the GPU. Also the number of parallel operations per stage is well balanced among the threads. The dependency on the structure of the tree is also removed. We suggest that this method is especially useful in applications where preorder or postorder traversal has been already done, and size of subtree is previously known. Examples of such algorithms include tree based algorithms, such as parallel biconnected components and ear decomposition.

6.1.1. Results of subtree queries. 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 need to perform a subtree query
- A small subset of nodes need to perform the query

These two are dealt with differently because coalescing will be much larger in the former. 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 per query time is given in the table below for different number of queries on a tree of size 20M. The queries are randomly chosen and are unsorted. For comparison with sequential implementation, a sequential expression evaluation algorithm, which performs a comparison of the key at the node to the minimum of subtree is done. CPU Time taken for a tree of 20 M nodes is 1 sec.

Size	100 log(n)	n/100	n/10
GPU time (ms)	.1	2	15

For complete queries, where the number of queries is equal to the number of nodes, the following table shows the result of :

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

6.1.2. Least common ancestor as DRS. Least common ancestor has a direct dual with DRS, LCA can be converted into a DRS and vice versa. Applications that map LCA to DRS have been shown in [9]. We strongly recommend using Euler tour technique as a preprocessor for tree algorithms especially in the context of the GPU. Given a general tree we perform level order and preorder traversal. Euler tour along with list ranking of Rehman *e*t. al [14] and the scan primitive [15] can be used to preprocess a given tree to generate the required traversal of the tree [12].Given an Euler tour ETT of a tree T arranged according to the rank of each edge. The LCA query can be answered by the following relation:

$$LCA[i,j] = I[RMQ_{ETT'}[R[i],L[j]]]$$
Here, $L[i]$=rank of edge $\langle parent[i],i\rangle$,
$R[i]$=rank of edge $\langle i,parent[i]\rangle$,
$ETT'[i] = levelorder[k]$: edge $\langle k,y\rangle = ETT^{-1}$[i].
$I[i] = k : edge\langle k,y\rangle = ETT^{-1}$[i].

A detailed commentary on ETT for the GPU is beyond the scope of the paper. In this paper, we assume that the arrays ETT, R,L and I are available. The relevant base arrays are created on the CPU and the results are moved on the GPU. The arrays are then processed for range queries. In our implementation, we have managed the divergence by the following method:

- All queries are loaded into the shared memory
- The queries are split into two, such that only one set requires RMQ.
- If in a thread block of size TB, let R queries require RMQ and TB-R do not. The threads which initially own the one of the TB-R query, processes the query, and then the remaining R queries are assigned to the first R threads. The assimilated results are then stored back on the global memory.

Results for LCA The times (independent of the ETT step) found for the $\frac{n}{10}$ random queries on a tree of size n (in millions) is given below:

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

A direct comparison with CPU implementations, in the absence of the timing data for ETT, is not done here.

6.2. Suffix array on the GPU

Suffix trees are the most prominent data structure for string queries. Suffix tree rely heavily on pointer jumping. In the context of the GPU, it is not an efficient practice. On the contrary, an array based schema is better suited. Suffix array [13] can thus be considered as a natural fit for the GPU, due to the locality of reference. A suffix array representation of a string typically consists of a suffix array, SA, containing starting indexes of the suffixes of the string in lexicographically sorted order.Additional supporting data structures assist computation, a few relevant data structures are Inverse

suffix array, SA^{-1} which contains the reverse indexes and LCP(Longest Common Prefix) contains the length of the maximum common prefix of adjacent elements in the suffix array. For example, for a string "SAAD“, the corresponding values in the suffix array is 2(AAD),3(AD),4(D),1(SAAD).

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

Addition of such metadata to the suffix array presents a new data structure called as enhanced suffix array [1]. Though additional application specific supporting data structures exist, for longest common extension, enhanced suffix array data structures suffice.

6.2.1. Longest common extension. Longest common extension of two strings is their longest common prefix. The longest common extension of a substrings starting from indices i,j is given by the following formula as suggested by Fishcher-Heun [9].

$LCE[i,j] = LCP[RMQ_{LCP}(SA^1[i]+1, SA^1[j])]$

Here LCE[i,j] is the longest common extension of string starting from two positions in the array. SA is an array storing the indices of suffixes of a string in sorted order. 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.

6.2.2. Results for Suffix Array. The suffix array is generated on the CPU and the LCP array is moved to the GPU. This 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 0.52 ms. The querying process is fundamentally RMQ, a detailed timing analysis of which has been already presented, hence a detailed analysis is spared here.

7. Conclusion and future work

We present first results on applicability of discrete range search on the GPU. Applicability of graph theory and string queries for the GPU are also studied. We propose that our results can be expanded to support tree based applications and string operations using the DRS primitives. We intend to utilize these primitives to make a set of general purpose applications amenable to the GPU.

References

[1] ABOUELHODA, M., KURTZ, S., AND OHLEBUSCH, E. *Replacing suffix trees with enhanced suffix arrays*, vol. 2(1),53Ű-86 of *J. Discrete Algorithms*. (2004).

[2] ALSTRUP, S., GAVOILLE, C., KAPLAN, H., AND RAUHE, T. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM, p. 264.

[3] ARROYUELO, D., CÁNOVAS, R., NAVARRO, G., AND SADAKANE, K. Succinct trees in practice. *Proc. 11th ALENEX* (2010).

[4] BENDER, M. A., PEMMASANI, G., SKIENA, S., AND SUMAZIN, P. Finding least common ancestors in directed acyclic graphs. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2001), Society for Industrial and Applied Mathematics, pp. 845–854.

[5] BERKMAN, O., SCHIEBER, B., AND VISHKIN, U. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms 14*, 3 (1993), 344–370.

[6] BERKMAN, O., AND VISHKIN, U. Recursive*-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science* (1989), pp. 196–202.

[7] CHEN, K., AND CHAO, K. On the range maximum-sum segment query problem. *Discrete Applied Mathematics 155*, 16 (2007), 2043–2052.

[8] FISCHER, J., AND HEUN, V. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. *Lecture Notes in Computer Science 4009* (2006), 36.

[9] FISCHER, J., AND HEUN, V. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. *Lecture Notes in Computer Science 4614* (2007), 459–470.

[10] GABOW, H., BENTLEY, J., AND TARJAN, R. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing* (1984), ACM New York, NY, USA, pp. 135–143.

[11] JACOBSON, G. Space-efficient static trees and graphs. In *In Proc. IEEE FOCS* (1989), p. 549Ű554.

[12] JÁJÁ, J. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992.

[13] MANBER, U., AND MYERS, E. Suffix arrays: A new method for on-line string searches. In *SIAM Journal of Computing* (1993), pp. 935–948.

[14] REHMAN, M., KOTHAPALLI, K., AND NARAYANAN, P. Fast and scalable list ranking on the GPU. In *Proceedings of the 23rd international conference on Conference on Supercomputing* (2009), ACM New York, NY, USA, pp. 235–243.

[15] SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2007), Eurographics Association, p. 106.