Parallel Computation of Standard Competition Rankings over a Sorted Array

Jingyuan Liang
Dept. of Elect. Eng. & Computer Sci.
Cleveland State University
Cleveland, OH, USA

Jonathan Bisnett
Dept. of Elect. Eng. & Computer Sci.
Cleveland State University
Cleveland, OH, USA

Alan Hylton
Space Communications and Navigation
NASA Glenn Research Center
Cleveland, OH, USA

Janche Sang
Dept. of Elect. Eng. & Computer Sci.
Cleveland State University
Cleveland, OH, USA

Chansu Yu
Dept. of Elect. Eng. & Computer Sci.
Cleveland State University
Cleveland, OH, USA

***Abstract*—The Standard Competition Ranking (SCR) is a commonly adopted ranking strategy and has been used in a wide range of applications, such as statistics, text mining, image processing, and so on. Though the sequential implementation of the SCR can be executed in linear time, it is not straightforward to design parallel algorithms for the SCR. In this paper, our focus is on the novel use of the parallel prefix computation method for calculating the SCR on a many-core Graphics Processing Unit (GPU). We also design a pthreads-based algorithm on a multi-core CPU which adopts a modified binary search to find the first item's rank in each partitioned segment. By integrating the modified binary search with the prefix computation, we later design and implement a more efficient hybrid algorithm on the GPU. The experimental results show that, as compared with the sequential execution on the CPU, our pthreads-based algorithm on a 12-core CPU can be roughly 8 times faster, while the hybrid algorithm on the GPU can achieve more than two orders of magnitude speedup.**

***Keywords*-Standard Competition Ranking, Parallel Prefix, Multi-core Computing, Many-core Computing, CUDA**

I. Introduction

Ranking refers to the data transformation by arranging a set of data items in numerical order and then assigning new values to denote where in the ordered set they are located. That is, the smallest data item is given the number 1, the second smallest item is given the number 2, the third smallest item is given the number 3, and so on. The numbers 1,2,3,$\cdots$ that are assigned to the various data items are called the ranks. Ranking is a popular approach to reducing complex information to a sequence of ordinal numbers. It has been frequently used in a wide range of applications, such as statistics, text mining, image processing, sports, etc. [1] [2].

Sometimes there are ties in the data items. This means that two or more data items are the same, so that there is no strictly increasing order. When this happens, the Standard Competition Ranking (SCR) method assigns the same rank number to the items that are equal and then leaves a gap in the rank numbers. The number of ranks which are left out in this gap is one less than the number of items that compared equal. As shown in the example below, the upper row is a sorted sequence of data items, while the lower row shows their corresponding rank numbers:

```
1.1  2.5  2.5  2.5  4.9  5.5  5.5  9.3
 1    2    2    2    5    6    6    8
```

There are three items with the value `2.5` and hence two rank numbers `3` and `4` are left out. Note that the ranks of all those items behind the tied items are unaffected. It can be seen in the example above, the items `4.9, 5.5, 5.5` and `9.3` are still ranked `5`th, `6`th, `6`th and `8`th, respectively. Furthermore, if an item is with the rank r, then there are $r-1$ items which are smaller than it. Because of these features, the SCR method is a commonly adopted ranking strategy.

Therefore, given an ordered set of n items, it is not difficult to find the rank of each item. Below shows the sequential code in C. Initially, the smallest item (i.e. `data[0]`) in the given sorted array is assigned the rank 1. Next, in the iteration loop, each data item is compared with its previous one. If equal, its rank number should be the same as the previous item's rank. Otherwise, its rank should be its array index plus 1 due to the array index in C starts from 0.

```
rank[0] = 1;
for (int i = 1; i < n; i++ ) {
   if(data[i] == data[i-1])
      rank[i] = rank[i-1];
   else
      rank[i] = i+1;
}
```

The sequential implementation of the SCR method can be executed in linear time $O(n)$, where n is the size of the ordered set. However, it is not straightforward to design parallel algorithms for computing SCR because the rank of an item depends on the relationship with its previous ones. Moreover, to the best of our knowledge, there is no parallel computing method for SCR. Hence, the goal of this paper is to design and implement efficient algorithms for computing SCR of an ordered set on the multi-core CPU and also on the many-core GPU platforms.

Recently, modern Graphics Processing Units have been increasingly adopted to accelerate the execution of applications by using hundreds or thousands of simple multi-threaded cores [3]. In NVIDIA GPU, a warp of 32 consecutive threads are bundled together to perform Single Instruction, Multiple Data (SIMD) parallel operations. To take advantage of the massively large data parallel computational power provided by a GPU, our focus is on the novel use of the parallel prefix computation method for calculating the SCR. Our implementation also uses the fancy shuffle functions which are supported in modern GPUs to make the execution even faster [4] [5]. These functions permit exchanging of variables (i.e. registers) between threads within a warp without using shared memory.

However, the parallel prefix approach is not suitable for being used on a CPU which has only a few or tens of cores available [6]. Hence, we design another new algorithm for multi-core CPU. Our idea is to partition the data into several segments and each segment is assigned for a pthread to handle. Each pthread utilizes a modified binary search to find the rank of the first item in the segment assigned, and then compute the ranks for the remaining items in the segment. By integrating this idea with the prefix computation, we later design a hybrid algorithm on the GPU to further improve the performance. The experimental results show that, as compared with the sequential execution on the CPU, our pthreads-based algorithm on the CPU can be approximately 8 times faster, while the hybrid algorithm on GPU can achieve at least two orders of magnitude speedup.

The organization of this paper is as follows. Section II describes the background and the related work. Section III goes into details of our algorithms using the multi-core and/or many-core environments. In Section IV, the experiments and the results for performance evaluation are presented. A short conclusion is given in Section V.

II. Background and Related Work

Given a set of n values $x_0, x_1, x_2, \ldots, x_{n-1}$ and a binary associative operator $\oplus$, the prefix computation is to generate the n quantities $s_0, s_1, s_2, \ldots, s_{n-1}$, where:

$$\begin{aligned} s_0 &= x_0 \\ s_1 &= x_0 \oplus x_1 \\ s_2 &= x_0 \oplus x_1 \oplus x_2 \\ &\vdots \\ s_{n-1} &= x_0 \oplus x_1 \oplus x_2 \cdots \oplus x_{n-1} \end{aligned}$$

For short, above can be rewritten as the chained relationship: $s_i = s_{i-1} \oplus x_i$, for i=1, 2, ..., n−1 with $s_0 = x_0$. Based on this relationship, a straightforward sequential algorithm for prefix computation simply traverses the input sequence, computing the different s_i, one after the other. For example, if $\oplus$ is an addition operation, the prefix computation on the input array of integers {3,1,0,2,4,3,5,2} would return the output {3,4,4,6,10,13,18,20}.

Note that the binary operator $\oplus$ in the prefix computation must be associative:

$$(x_0 \oplus x_1) \oplus x_2 = x_0 \oplus (x_1 \oplus x_2) = x_0 \oplus x_1 \oplus x_2$$

Namely, in order to get the number s_2, it is not necessary to compute $x_0 \oplus x_1$ first. Therefore, the original parallel prefix approach utilizes this property to perform the operations in parallel. Assuming the number of processors is the same as the input data size n, where n is power of 2, the parallel algorithm consists of $\log n$ iterations. During each step j, there are $n-2^j$ processors which perform the binary operation $\oplus$ concurrently and the indices of the two elements accessed by a processor are separated by 2^j, as shown in the pseudo code in Figure 1.

```
for $j = 0$ to $\log n - 1$
    for $i = 2^j$ to $n-1$ in parallel
        $s_i = s_{i-2^j} \oplus s_i$
    endfor
endfor
```

Fig. 1. Pseudo code of the Parallel Prefix Approach

Parallel prefix is an important technique that has been frequently used to parallelize seemingly sequential operations, typically in $O(\log n)$ time [6]. As shown in Figure 2 below, it only needs 3 iterations to compute a list of 8 values. Parallel prefix has a wide range of applicability in science and engineering. One application is solving the list ranking problem [7]. That is, given a singly linked list, find the location of each node in the list −− specifically, its distance from the end of the list. By using parallel prefix as well as pointer jumping, every list element's position can be correctly determined in a logarithmic number of steps [8]. Another interesting application is to simulate the First Come First Served G/G/1 queuing network [9]. The formulas for calculating the arrival and departure times have been transformed into the linear recurrence relations which can be computed efficiently using parallel prefix.

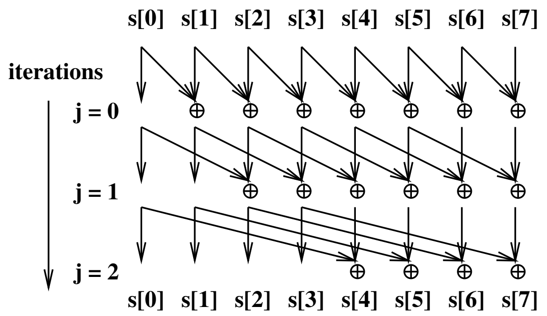
s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7]
iterations
j = 0
j = 1
j = 2
s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7]

Fig. 2. Parallel Prefix Computation

The parallel prefix computation is appropriate for SIMD parallel computers because its fine-grained parallelism comes from simultaneous operations across large amounts of data, rather than from multiple threads of control. With the advance of hardware technology, a GPU can have hundreds or even thousands of processing cores. Therefore, it has been used in recent years for improving the performance of various computational intensive applications [10] [11]. It consists of a scalable number of streaming multiprocessors (SMs) and each SM contains a group of streaming processors (SPs) [12]. The kernel function, which is executed on the device, is composed of a grid of threads to be executed on the SPs. More precisely, a grid is divided into a set of blocks and each block contains multiple warps of threads. Blocks are distributed evenly to different SMs to run. A warp has 32 consecutive threads and each thread's lane ID is its index within a warp, ranging from 0 to 31. The GPU device has its own off-chip device memory (i.e. global memory) and hence data needs to be transferred from the host CPU before executing the kernel function. Furthermore, threads can access the fast on-chip memory resources, such as shared memory and registers. These are per-block resources and will be released when all the threads of the same block finish their executions.

Note that the new shuffle functions, which are available on the Kepler and later GPUs (Compute Capability 3.0 and above), allow threads within the same warp to read each other's registers. Using the function __shfl_sync(unsigned mask, int v, int srcLane) as an example, the caller thread will get the value of the variable v held by the thread with lane ID srcLane. It behaves the same as broadcasting if every thread in the warp copies from the same source lane. For another example, the function __shfl_up_sync(unsigned mask, int v, int d) will let the lane k thread read the variable v held by the lane $(k-d)$ thread.

The __ballot_sync(unsigned mask, int p) intrinsic function returns a 32-bit integer in which bit k is set if and only if the predicate p provided by the thread with lane ID k is non-zero. Note that the lane ID is the thread's index within a warp, ranging from 0 to 31. In other words, the __ballot_sync() function collects the predicates from all threads in a warp into a 32-bit integer and returns this integer to every thread. It has been used to accelerate the stream compaction task [13]. Another two intrinsic functions __brev(unsigned int x) and __ffs(int x) can be used together to examine the ballots. The former reverses the bit order of an unsigned integer, while the latter finds the position of the least significant bit set to one in a 32-bit integer.

III. New Parallel Algorithms

As mentioned in the earlier section, the calculation of the SCR seems inherently sequential. However, we can restructure the problem so that it can be carried out by using the parallel prefix approach. For each element x_i, $0 \le i < n$, it holds a 2-tuple (d_i, r_i), where d_i is its datum from the input sorted array d and r_i is its rank which is initialized to be $i+1$. The binary operator $\oplus$, which is applied to x_i and x_j, $i < j$, is defined as follows:

$$\begin{aligned} & x_i \oplus x_j \\ = \; & (d_i, r_i) \oplus (d_j, r_j) \\ = \; & \texttt{if } d_j == d_i \texttt{ return } (d_j, r_i) \\ & \texttt{else return } (d_j, r_j) \end{aligned}$$

More precisely, the operator $\oplus$ compares the input data elements d_i and d_j, $i < j$. If they are equal, the rank of the element d_j will be the same as the rank of d_i. Otherwise, its rank remains unchanged. In order to adopt the parallel prefix method, we have to show that the binary operator $\oplus$ which is defined above still has the associative property. That is, repeated application of the binary operation $\oplus$ produces the same result regardless of how pairs of parentheses are inserted in the expression. Table I shows all possible cases for calculating $x_a \oplus x_b \oplus x_c$, where $a < b < c$ in detail. For example, if $d_a = d_b = d_c$, then the calculations by grouping x_a and x_b, first

$$\begin{aligned} & (x_a \oplus x_b) \oplus x_c \\ & = (d_b, r_a) \oplus (d_c, r_c) \\ & = (d_c, r_a) \end{aligned}$$

and by grouping x_b and x_c, first

$$\begin{aligned} & x_a \oplus (x_b \oplus x_c) \\ & = (d_a, r_a) \oplus (d_c, r_b) \\ & = (d_c, r_a) \end{aligned}$$

yield the same result.

A. Naive Parallel Prefix Algorithm

Our first implementation is a modification to the original parallel prefix method illustrated in Figure 2. In our modified algorithm, the ordered array d stores the input data and the corresponding rank for each element in d will be calculated in the output array r. We firstly initialize in parallel each element of the result array to be its index plus 1. During the iteration steps, the rank copying operation is performed only when the current data element equals to the data element at the index of the copying source. Otherwise, the rank remains unchanged. Figure 3 shows the pseudo code of the modified ranking algorithm.

The major downside of this algorithm (inherited from the original parallel prefix algorithm) is that each iteration of the outer sequential for-loop requires launching a kernel function on GPU and hence produces more synchronization overhead. Also in CUDA, it needs to access the whole memory space multiple times, for both data elements and ranking elements. Because of the design of CUDA architecture, these memory accesses generate a major performance penalty. However, it is a good demonstration showing how the parallel prefix operations can be used for ranking.

TABLE I
THE ASSOCIATIVE PROPERTY OF $x_a \oplus x_b \oplus x_c$, WHERE $a < b < c$

case	$x_a \oplus x_b$	$(x_a \oplus x_b) \oplus x_c$	$x_b \oplus x_c$	$x_a \oplus (x_b \oplus x_c)$
$d_a = d_b = d_c$	(d_b, r_a)	$(d_b, r_a) \oplus (d_c, r_c) = (d_c, r_a)$	(d_c, r_b)	$(d_a, r_a) \oplus (d_c, r_b) = (d_c, r_a)$
$d_a = d_b < d_c$	(d_b, r_a)	$(d_b, r_a) \oplus (d_c, r_c) = (d_c, r_c)$	(d_c, r_c)	$(d_a, r_a) \oplus (d_c, r_c) = (d_c, r_c)$
$d_a < d_b = d_c$	(d_b, r_b)	$(d_b, r_b) \oplus (d_c, r_c) = (d_c, r_b)$	(d_c, r_b)	$(d_a, r_a) \oplus (d_c, r_b) = (d_c, r_b)$
$d_a < d_b < d_c$	(d_b, r_b)	$(d_b, r_b) \oplus (d_c, r_c) = (d_c, r_c)$	(d_c, r_c)	$(d_a, r_a) \oplus (d_c, r_c) = (d_c, r_c)$

```
for $i = 0$ to $n-1$ in parallel
    $r_i = i + 1$
endfor

for $j = 0$ to $\log n$ - 1
    for $i = 2^j$ to $n-1$ in parallel
        if $d_i$ == $d_{i-2^j}$
            $r_i = r_{i-2^j}$
    endfor
endfor
```

Fig. 3. The Naive Parallel Prefix approach for computing SCR

B. Shuffle Scan

NVIDIA included a prefix sum implementation based on shfl_scan in CUDA examples [14]. In a similar but slightly improved manner, we adapted it to become a ranking algorithm and evaluated this algorithm.

First, we do similar initialization for the ranking elements. Instead of a simple index as the ranking value, we are making them a tuple of the ranking value and a flag, where the flag indicates whether the data element at this index is equal to the previous data element or not. This eliminates accesses to the data elements in all following steps. Practically, considering there cannot be more than 2^{30} elements due to memory limitation here, we combine the ranking value and the flag into one 32-bit integer by using the lower 31 bits to store the ranking value and the highest bit for the flag.

Second, we change the addition operation to become either a copying operation from a previous value or a no-op depending on the flag. That is, it is a copying (returns left operand) if the right operand does not have the flag set (meaning the data element here equals the previous one), or is a no-op (returns right operand) if the right operand has the flag set (meaning the data element here does not equal the previous one).

The final change is just to extend the prefix sum operation to be able to operate on any array size, subject to memory limit. The original algorithm only works on 65536 elements but this change is not directly connected to ranking.

```
float pelm = __shfl_up_sync(0xffffffff, elm, 1);

unsigned int votes = __ballot_sync(0xffffffff, elm!=pelm);

unsigned int votes_rev = __brev(votes&((1<<(laneID+1))-1));

rank = belm_idx + ((32-__ffs(votes_rev)) & 0x1f);
```

Fig. 4. Finding Ranks via Intrinsic Functions

C. Shuffle and Intrinsic Functions with Modified Binary Search

This algorithm starts by establishing the idea of a global warp which in this context is a block of 1024 elements that will all be processed by a single warp in an iterative process. Naturally, during the processing if the global warp id is zero, this is the first group of values and would start with the ranking of 1. This can be calculated without having to retrieve a value from a previous group.

Assuming this is not the first global warp, we need to get a beginning rank for the group. We take the 32 values at the end of the prior global warp and then use a single shuffle-up by one position that allows each thread to determine if its value is different from its predecessor, as shown in Figure 4. This in turn sets a value of 1 for a difference and a 0 for the same. The intrinsic function (`__ballot_sync`) takes the one or zero from each thread and combines them into an unsigned 32-bit integer. This unsigned integer directly reflects where the values have changed in the block of 32 threads. To find the bits that have changed, the intrinsic function `__ffs` can be used, but unfortunately the order of the bits from our threads is in the wrong order. To correct this short coming, the intrinsic function `__brev` is used to reverse the bits in the ballot integer (MSB <-> LSB). Now that the bits are in the order necessary to use `__ffs`, a mask is generated that will only expose the bits for values in threads below the current thread position. The value returned by the `__ffs` function can then be converted back to determine the thread below the current one that changed. The position can then be used to set the ranking for the current thread.

Once this process is complete, if the rank for the last value in the group of 32 is different from the position of the first value in the group, a change occurred within the 32 values and the rank for the last one is the beginning rank for the current global warp. If the values are the same, the code then moves into a modified binary search working backwards through the

```
belm = d[brank];
int low = 0;
int high = brank -1 ;

while (low <= high) {
   int m = (low+high) / 2 ;

   if (d[m] == belm) { // check first half
       high = m - 1;
       brank = m;
   }
   else { // check second half; brank remains the same
       low = m + 1;
   }
}
```

Fig. 5. Modified Binary Search

values until it finds the first occurrence of the current value. For clarification purposes, it should be noted that the modified binary search continues until the range shrinks to empty, as shown in Figure 5 . This assures that we have found the first occurrence of the value we are seeking. The code can then take that position within the array as the starting rank for the current global warp.

Now that the beginning rank for the current global warp is known, the code performs a loop of 32 iterations with each iteration doing the shuffle-ballot-brev-ffs process for each 32 values then advancing 32 positions and repeating the process again until the entire 1024-value block is complete.

The advantage of this algorithm is that only a single run through the entire array is necessary to calculate all the rankings. The modified binary search allows for a single kernel to do all the ranking rather than requiring several kernels to pass over the array and shuffle up the values. The disadvantage is that the range of the modified binary search becomes larger as the processing moves closer and closer to the end of the array.

D. Pthreads with Modified Binary Search on CPU

As mentioned before, it is not suitable to implement parallel prefix on a CPU which has only a few or tens of cores available. Hence, we need to design a different algorithm for multi-core CPU. In this method, the array is broken up and spread across several independent threads. Therefore, each thread must find the beginning rank for its block of values. The modified binary search, as described previously, works backwards through the values prior to the current value block, cutting the size in half each time and seeking the first occurrence of the current value. Also as noted previously, this is a modified search and will continue until the range is empty.

Once the value is found, the code continues as for the sequential processing above and marches through the value block setting the rankings based on the current value compared to the preceding value.

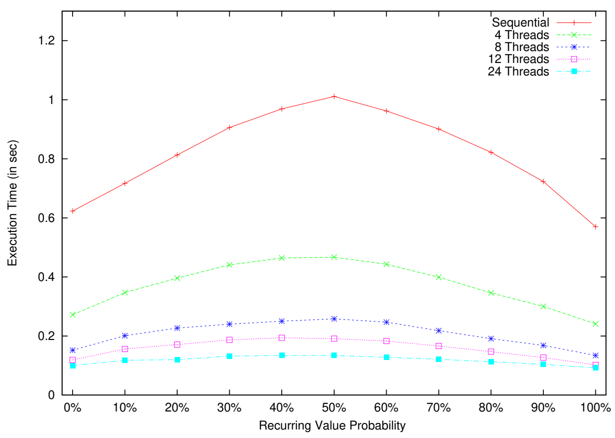

Fig. 6. Execution Time versus Probability for CPU pthreads

IV. Experimental Results

The experiments were conducted on a high-end workstation sponsored by NASA GRC. This machine, running the Ubuntu 18.04, has a Xeon Silver 4116 CPU (12 cores, 24 threads) clocked at 3.0 GHz and has 64 GB in total of Registered ECC server-grade memory. The GPU device used in this computing platform was the NVIDIA GeForce RTX 2080 Ti, which is built with evolutionary NVIDIA Turing architecture [15] and contains 68 streaming multiprocessors (4352 CUDA cores in total), 11GB GDDR6 memory and 1.65 GHz GPU clock rate.

The ordered values for ranking are generated using a floating-point recurring probability from 0 to 1 such that 0 will cause the values to all be different and 1 will cause the values to all be the same. This allows us to see how each algorithm behaves based on different distributions of change within the array of values.

Figure 6 shows the execution timing results gathered when running the CPU pthreads code against 128 million elements with probabilities ranging from 0 to 1 in intervals of 0.1. Each execution timing is the result of 10 executions and extraction of the lowest execution time. It is important to note that the evaluation server used was not dedicated entirely to this evaluation and could have had other activity that may have had an impact on the performance. Thus, the reason for multiple executions and taking the lowest value, was to minimize other factors. A review of these numbers shows that as the number of threads increase, the execution time drops. Not shown in the figure is that if we use more than 24 threads, the execution time will not drop further. This makes sense, since the machine used for the evaluation has 12 cores with 24 hyper threads.

We also ran the experiment to compare the three approaches we proposed on the GPU platform. The timing results can be found in Figure 7. Among the three, the naive approach performs the slowest because it needs to launch the kernel function on GPU $\log(n)$ times and each kernel invocation

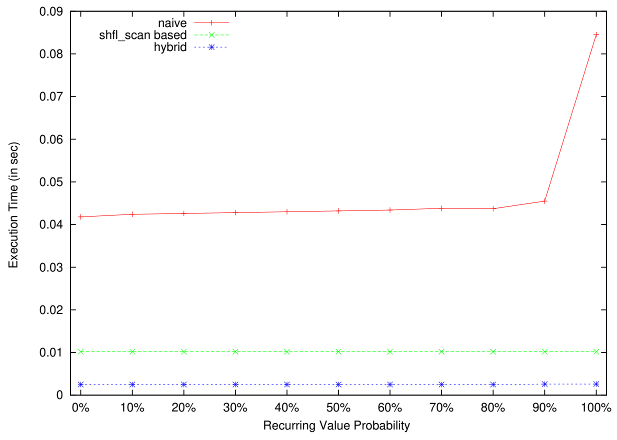

Fig. 7. Execution Time versus Probability for different algorithms on GPU

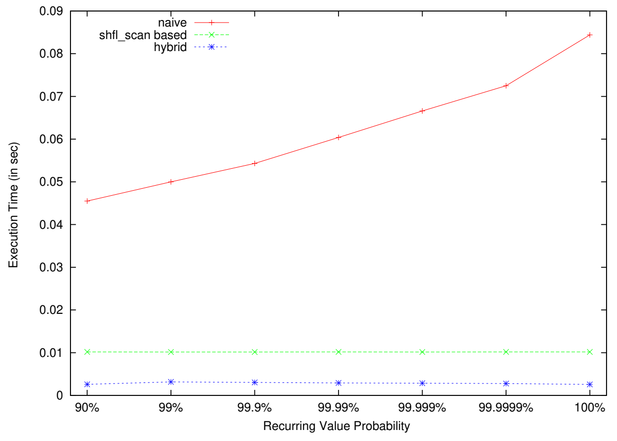

Fig. 8. Execution Time versus High Recurring Probability for different algorithms on GPU

will access the input and output arrays, while the hybrid method which utilizes the modified binary search and intrinsic functions performs the fastest because it only requires one kernel function call. Note that these three approaches perform much faster as compared with the timing results on the CPU in Figure 6,

Another evaluation, as shown in Figure 8, is like the previous one but focuses on the upper end of the recurring value probability where the likelihood of the value being the same as the previous value is much higher. This can present some difficulties to the various algorithms, because the beginning ranking for a group of values is not necessarily found near the current value. This requires some special processing to work backwards to find the proper beginning rank for the group. This evaluation also uses 128 million values but looks at the high end of the probability range starting at 0.9, advancing to 0.99, then 0.999, and so on until 0.999999 and finally 1.0.

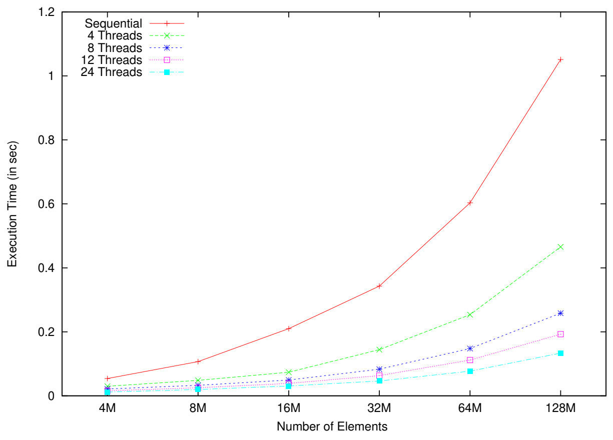

Fig. 9. Execution Time versus Number of Elements for CPU Pthreads

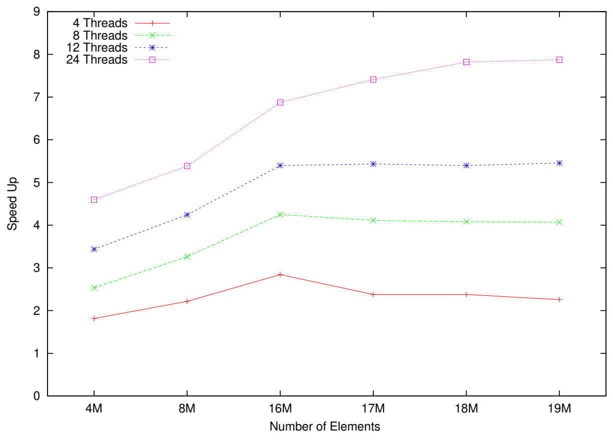

Fig. 10. Speed Up for CPU Pthreads

In another experiment, we measured the pthreads-based algorithm performance by varying the input from 4 million values to 128 million values, while using the recurring value probability of 50%. This allows us to see if an algorithm performs better or worse based on the size or number of values being ranked. Figure 9 depicts the results running on the CPU. The corresponding speed-up ratios, which are the ratios of the sequential time to the parallel time, are shown in Figure 10. It can be observed that using 24 pthreads can be roughly 8 times faster than the serial execution on the CPU.

A similar experiment was also conducted for the three parallel prefix-based approaches on the GPU device. Figure 11 and Figure 12 show the timing results and the speed up ratios, respectively. It can be seen that our hybrid approach can be more than 400 times faster than running sequentially on the CPU.

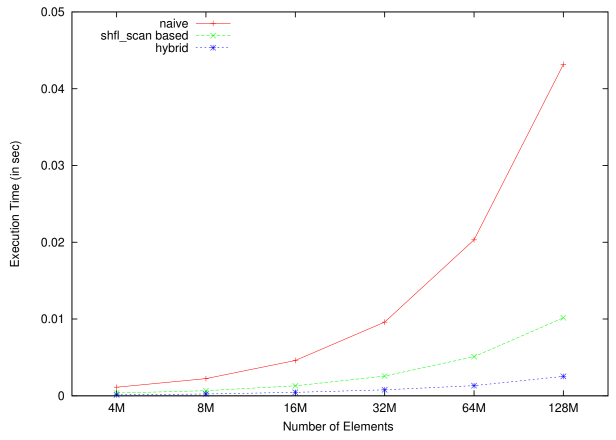

Fig. 11. Execution Time versus Number of Elements for algorithms on GPU

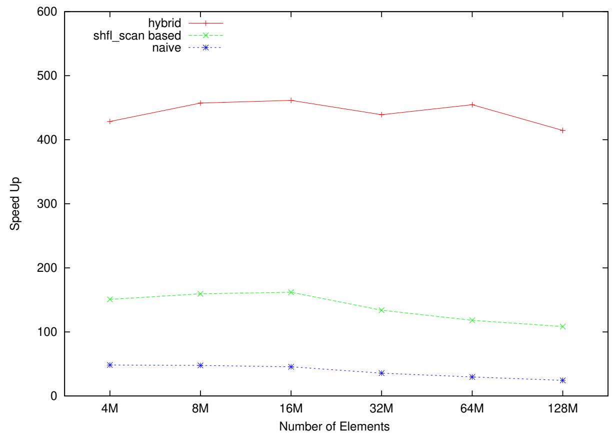

Fig. 12. Speed Up for algorithms on GPU

V. Conclusion and Future Work

The computation of SCR over a sorted array seems inherently sequential because the rank of an item depends on the relationship with its previous ones. We restructured the problem and introduce a new associative binary operation so that the SCR calculation can be carried out by using the parallel prefix approach. We presented three implementations on the GPU many-core computing platform and one parallel implementation on the CPU multi-core environment. The experimental results show that the pthreads-based approach on a 12-core CPU can achieve almost 8 times faster than the sequential execution. The results on the GPU platform are even more encouraging. The hybrid scheme which exploits the warp shuffle and some intrinsic functions as well as the modified binary search method, can achieve more than two orders of magnitude speedup relative to a serial CPU implementation. Furthermore, we have extended our ideas to implementing other ranking strategies, such as Modified Competition Ranking, Fractional Ranking, etc. [16] and currently are conducting experiments to evaluate their performance.

Acknowledgment

This research was supported in part by the Summer Faculty Fellowship and the equipment sponsorship from the NASA Glenn Research Center.

References

[1] H. A. David and H. N. Nagaraja, *Order Statistics*, 3rd ed. Wiley, 2003.

[2] G. Heygster, “Rank filters in digital image processing,” *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 148–164, 1982.

[3] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016.

[4] M. Harris, “CUDA Pro Tip: Do The Kepler Shuffle, PARALLEL FORALL,” http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/, 2015.

[5] L. S. N. Nunes, J. L. Bordim, K. Nakano, and Y. Ito, “A Memory-Access-Efficient Implementation of the Approximate String Matching Algorithm on GPU,” in *Proceedings of International International Symposium on Computing and Networking (CANDAR)*, 2016.

[6] W. Hillis and G. Steele, “Data Parallel Algorithms,” *Comm. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[7] J. C. Wyllie, “The Complexity of Parallel Computations,” PhD thesis, Cornell University, Ithaca, NY, USA, Tech. Rep., 1979.

[8] M. J. Quinn, *Parallel Computing: Theory and Practice*, 2nd ed. McGraw-Hill, 1994.

[9] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani, “Algorithms for unboundedly parallel simulations,” *ACM Trans. on Computer Systems*, vol. 9, no. 3, pp. 201–221, Aug. 1991.

[10] A. Hylton, G. Henselman-Petrusek, J. Sang, and R. Short, “Tuning the performance of a computational persistent homology package,” *Software: Practice and Experience*, vol. 49, no. 5, pp. 885–905, May 2019.

[11] J. Sang, C. Lee, V. Rego, and C. King, “Experiences with implementing parallel discrete-event simulation on GPU,” *Journal of Supercomputing*, vol. 75, pp. 4132–4149, Aug. 2019.

[12] NVIDIA, *CUDA Programming Guide version 10.0* , 2018.

[13] V. Rego, J. Sang, and C. Yu, “A Fast Hybrid Approach for Stream Compaction on GPUs,” in *Proceedings of International Workshop on GPU Computing and Applications*, 2016.

[14] NVIDIA, “CUDA Parallel Prefix Sum with Shuffle Intrinsics (SHFL_Scan),” http://docs.nvidia.com/cuda/cuda-samples/index.html.

[15] E. Kilgariff and H. Moreton and N. Stam and B. Bell , “NVIDIA Turing Architecture In-Depth,” https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/, 2018.

[16] Wikipedia, “Ranking,” https://en.wikipedia.org/wiki/Ranking.