Parallel version of the Generalized Multidimensional K-Maximum Subarray Problem with CUDA-based Implementation

A.Geetha Venkatesh
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
agv.bits@gmail.com

***Abstract*—The Maximum Subarray Problem (MSP) finds the segment of an array that has the maximum summation over all other combinations. The D-dimensional K-Maximum Subarray Problem is a generalized version of MSP, which finds K maximum subarrays in a D-dimensional array. The main aim of this course project is to develop a parallel version for this generalized version of the MSP. This work involves analyzing a sequential lower bounded, O(N+K) algorithm for K-MSP in 1-D and then developing a CUDA-based implementation which is competitive with the sequential version. Even though the problem is highly sequential, the CUDA version achieves more than 13X of speed-up in performance compared to the sequential for an array size of 1024^2, 1-D and K=50. This parallel version is also generalized and extended to D-dimensional problem.**

***Index Terms*—Maximum Subarray, Persistent Data Structure, Prefix Sum.**

I. Introduction

The Maximum Subarray Problem (MSP) problem is concerned about retrieving a segment of consecutive array elements that has the maximum summation over all the other possible combinations in this array. If the array elements are of only positive numbers, then the maximum summation in such case will be the entire array; therefore the MSP always deals with arrays of positive and negative numbers. For example, consider the sequence of values −3, 1, −2, 4, −1, 2, 1, −5, 3; the contiguous subarray with the largest sum is 4, −1, 2, 1 with sum 6. i.e, The maximal sub-vector of an array A with n numbers is the sub-vector A[i, . . . , j] maximizing $\sum_{s=i}^{j} A[s]$. Finding the K sub-vectors with the largest sums is a natural extension of this and is known as the K maximum subarray problem. The problem can be extended to any number of dimensions.

This problem has significant projection in several applications in the medical field such as genomic sequence analysis at which it predicts the membrane topology of proteins leading to understanding their functions; which is an essential step for the development of antibodies and drugs. In computer vision, this problem is employed in finding the brightest region of an image; which can be used later for image analysis. As a practical application in the medical field, we use this technique in the work for the detection of macrocalcifications in mammographic images [1].

II. Related Work

The Maximum Subarray Problem was first posed by Ulf Grenander of Brown University in 1977, as a simplified model for maximum likelihood estimation of patterns in digitized images. A linear time algorithm was found soon afterwards by Jay Kadane of Carnegie-Mellon University (Bentley 1984). Then afterwards in literature, there are a couple of linear order sequential algorithms developed for solving the MSP. Similar problem posed for higher dimensional arrays has solutions which are more complicated; e.g., Takaoka (2002).

The lower bound for the K maximal subarray problem is $\Omega(N+K)$ in 1-D, since an adversary can force any algorithm to look at each of the N input elements and the output size is $\Omega(K)$. Brodal & Jørgensen (2007) showed how to find the K largest subarray sums in a one-dimensional array, in the optimal time bound O(N + K) [2].

The recent work on CUDA parallel version of the MSP problem is for 2D problem, is done by Salah Saleh group [3], which is a Dec-2012 publication. This indicates that the generalized parallel version of MSP problem extended for K-maximums and for D-dimensions is unsolved for GPU architecture using CUDA-based implementation. This is an initial attempt to solve this problem using the GPU architecture, to develop CUDA based implementation for K-maximum subarray problem and extending it to the generalized version of D-dimensions.

III. Methodology

A. Brodal & Jørgensen Sequential Method

The algorithm followed is complex and the key main points are tried to be summarized here. In this algorithm the term heap denotes a max-heap ordered binary tree. The basic idea of this algorithm is to build a heap storing the sums of all ${}^{n}C_2 + n$ sub-vectors and then use Frederickson's binary heap selection algorithm to find the K largest elements in the heap.

For the heap formation, grouping is done based on the triples that end with same index from the ${}^{n}C_2{+}n$ sums. The suffix set of triples corresponding to all sub-vectors ending at position j is denoted by Q^j_{suf}, and this is the set $\{(i, j, sum)|1 \leqslant i \leqslant j \ni sum = \sum_{s=i}^{j} A[s]\}$. The Q^j_{suf} sets can be incrementally defined as follows:

$$Q^j_{suf} = \{(j, j, A[j])\} \cup \{(i, j, s + A[j])|(i, j{-}1, s) \in Q^{j-1}_{suf}\}. \quad (1)$$

As stated in equation (1) the suffix set consists of all suffix sums in Q^{j-1}_{suf} with the element A[j] added, as well as the single element suffix sum A[j].

Using this definition, the set of triples corresponding to all ${}^nC_2 + n$ sums in the input array is the union of the n disjoint Q^j_{suf} sets. These Q^j_{suf} sets are represented as heaps and is denoted by H^j_{suf}. Assuming that for each suffix set Q^j_{suf}, a heap H^j_{suf} representing it has been build, a complete binary heap H on the top of all these heaps is constructed. The keys for the N−1 top elements is set to ∞ (see Figure 1). To find the K largest elements, we extract the N−1 +K largest elements in H using the binary heap selection algorithm of Frederickson and discard the N−1 elements equal to ∞.

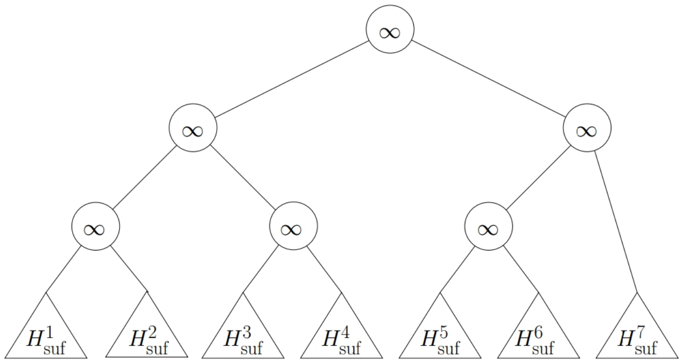


Fig. 1. Example of a complete heap H constructed on top of the H^j_{suf} heaps. The input size is 7.

Since the suffix sets contain $\Theta(n^2)$ elements the time and space required is still $\Theta(n^2)$ if they are represented explicitly. The algorithm uses persistent data structures for constructing heaps.

Data structures are usually ephemeral, meaning that an update to the data structure destroys the old version, leaving only the new version available for use. An update changes a pointer or a field value in a node. Persistent data structures allow access to any version old or new. Partially persistent data structures allow updates to the newest version, whereas fully persistent data structures allow updates to any version [4].

The main steps, algorithms and data structures used for optimization to make the algorithm to run in linear order are specified here.

B. Steps For Parallelization

1. Creating a persistent data structure which can be accessed in parallel and updated in parallel. Because using other data

Algorithm 1 A Linear Time Sequential Algorithm for the K Maximal Sums Problem

1. To obtain a linear time construction of the heaps H^j_{suf}, an implicit representation of a heap, called Iheap that contains all the sums is constructed in linear time.
   a. The Iheaps are designed to have the partial persistence.
   b. The Iheap data structure is for representing the Q^j_{suf} sets that supports insertions in amortized constant time.
2. Construction of these Iheaps for the N elements completes in O(N) time.
   a. According to equation(1), the set Q^{j+1}_{suf} can be constructed from Q^j_{suf} by adding A[i+1] to all elements in Q^j_{suf} and then inserting an element representing A[i+1].
   b. To avoid adding A[i+1] to each element in Q^j_{suf}, each Q^{j+1}_{suf} set is represented as a pair $\langle \delta_j, H^j_{suf}\rangle$, where H^j_{suf} is a version of a partial persistent Iheap containing all sums of Q^j_{suf} and δ_j is an element that must be added to all elements.
   c. With this representation a constant can be added to all elements in a heap implicitly by setting the corresponding δ.
   d. The following is the incremental construction of the pair $\langle \delta_{j+1}, H^{j+1}_{suf}\rangle$:
   $$\langle \delta_0, H^0_{suf}\rangle = \langle 0, \emptyset\rangle \quad (2)$$
   $$\langle \delta_{j+1}, H^{j+1}_{suf}\rangle = \langle \delta_j + A[i+1], H^j_{suf} \cup \{-\delta_j\}\rangle \quad (3)$$
   e. Using the node copying technique [4] build H^{j+1}_{suf} from H^j_{suf} in O(1) time amortized without destroying H^j_{suf}
   f. Time for constructing the N pairs $\langle \delta_j, H^j_{suf}\rangle$ is O(N).
3. A complete binary tree is constructed over these Iheaps and using Frederickson Heap Selection Algorithm [5] the N−1 +K largest elements in a heap are found in O(N−1 +K) time.
   a) The main part of the algorithm is for locating an element 'e', with $K \leqslant rank(e) \leqslant cK$ for some constant c.
   b) After this element is found, the heap is traversed and all elements larger than 'e' are extracted.
   c) Standard selection is then used to obtain the K largest elements from these O(K) extracted elements.
4. Using reduction repeatedly d-dimensional arrays are reduced to one dimensional and the Iheaps formed (by 1,2,3 steps) from these arrays are all considered, for the complete binary heap H construction.
   a. For example: in 2-D, for all i, j with $1 \leqslant i \leqslant j \leqslant$ N we take the sub-matrix consisting of the rows from i to j and sum each column into a single entrance of an array. The array containing the rows from i to j can be constructed in O(N) time from the array containing the rows from i to j−1.
   b. For d-dimension problem, the algorithm uses the reduction to the d−1 -dimensional case by constructing ${}^{N_d}C_2 + N_d$ d−1 -dimensional problems.i.e, ${}^NC_2 + N$ 1-D arrays for 2-D problem.

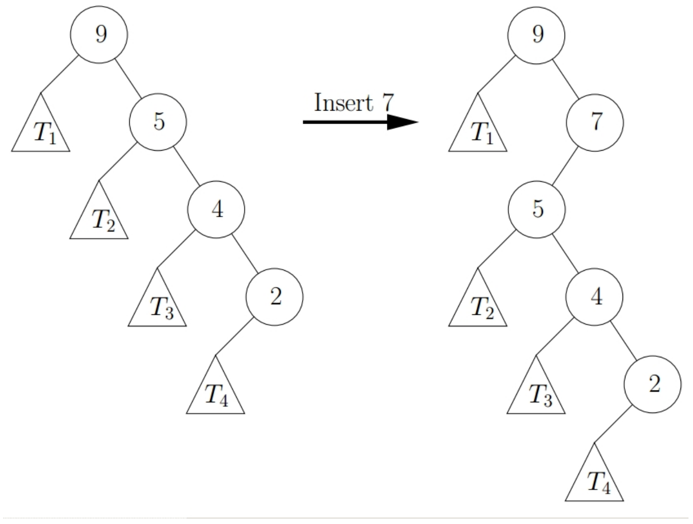


Fig. 2. An example of an insertion in the Iheap. The element 7 is compared to 2, 4 and 5 in that order, and these elements are then removed from the rightmost path.

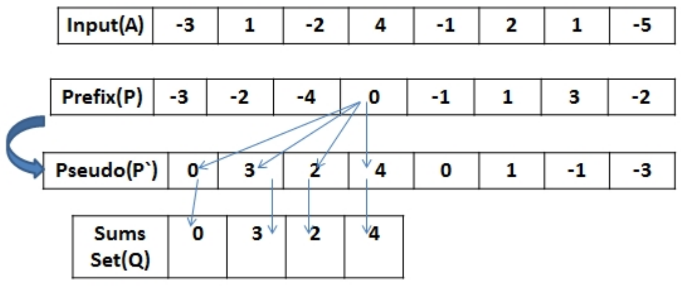
structures increases the time to be more than O(N).

2. The construction of the N Iheaps (if constructed) need to be parallelized, such that it does not exceed O(N).
3. After the $\langle \delta_j, H^j_{suf} \rangle$ pairs construction, the element 'e' of Frederickson Heap Selection Algorithm need to be found in less than O(N−1 +K) time.
4. The final Standard selection should be from O(K) elements.
5. Creating the combination data for multidimensional.

C. Parallel Implementation

For handling the persistent data structure which can be accessed in parallel and updated in parallel.Used simple array data structure itself, and this array can give all the elements of the Q^j_{suf} set. Construct the prefix sum array, say P with N elements for all the elements in the array A. And we have

$$Q^j_{suf} = \{(P[j]+P'[i]) | (i <= j) \& (P'[0] = 0, P'[i] = -P[i-1])\}. \quad (4)$$



Input(A) -3 1 -2 4 -1 2 1 -5
Prefix(P) -3 -2 -4 0 -1 1 3 -2
Pseudo(P`) 0 3 2 4 0 1 -1 -3
Sums Set(Q) 0 3 2 4

Fig. 3. An example for the Prefix Sum array being used for get the Q^3_{suf}

As shown in the example from Fig.3, the computed Prefix sum array itself used as a persistent data structure to get the Q^j_{suf} set.

Skipped the step of constructing the heaps and instead, just found the maximum value in P' till that element sequentially. As this step has high load imbalance, initial thread gets 1 element load whereas the final thread gets N elements load and as CPU is faster than GPU for a single thread, used CPU to get the maximum array in O(N).

Parallelized finding the element 'e' of Frederickson Heap Selection Algorithm. Initially found the maximum with in the Blocks(each block has 512 threads and 1024 elements) and collected them. And then out of these collected Block maximums, selected the K^{th} maximum as element 'e1'. Also collected the individual K^{th} ranked element at each Block, say 'e2'. Now chosen the $\max\{e1, e2\}$ as the threshold element 'e' for each block. As this is done in parallel the time taken for this step is less than O(N).

After the threshold value 'e' is found for each individual Block of threads, the threads which have their maximum value greater than this threshold find their subset elements which are greater than this threshold. All these elements are collected into the global memory and the largest K elements of these is found by the next kernel call. If the problem is multidimensional the K largest found in the previous combination array are also added to this set for the updated largest. And as this step is also parallelized and O(K) elements are dealed with, time is saved here. Each thread of the CUDA code deals with one element of the array, and this thread finds the greater than threshold value elements from its Q^j_{suf} set, which takes at most O(N)time. The gained O(N) time of sequential heap forming step is used here.

In sequential, creating the combination data for multidimensional did not took much time when compared to other steps, and due to time constraints did not parallelize this step. Even this step can be parallelized, but there needs a lot of data movement between GPU and CPU, which may result in no better time saving.

IV. Experiments and Results

A. Optimization Experiments

For forming the persistent data structure which can be parallelized, experimented with heaps data struture similar to the sequential version. But as the complexity kept on increasing, switched to the simple arrays and used the concept of persistent data structure of the pair $\langle \delta_j, H^j_{suf} \rangle$ to take the prefix sum array and was able to construct the Q^j_{suf} successfully when required.

Took the concept of finding the threshold element 'e' as in Frederickson Heap Selection Algorithm and formed a better threshold value 'e', considering both the Block level K^{th} rank and global K^{th} rank elements. Doing this way reduced the number of threads, that need to search through all its sum values for finding the values which are grater than the threshold value 'e'.

After the sum values from the individual threads are collected, invoked one more kernel in which each thread handles one element and finds if its value is ranked more than K or not, largest sum is ranked 0. Here sorting of all the elements is not being done and each thread counts the values greater than it, if it is more than K the thread terminates.

Threads are maintained in warps and ensured that consecutive threads deal with consecutive elements, so that cuda memory coalescing is achieved. And during the shared memory access, used in loops [(i+subthreadid)%NumOfElements] so that memory bank conflicts does not occur. Also tried for avoiding the warp divergence, but at the point when the threads needs to find their respective threshold exceeding values the warp divergence occur because of load imbalance.

B. Results

Even though the problem is highly sequential, successfully parallelized few steps of the algorithm taken and achieved a good speedup of upto 13X. The speedup are calculated maintaining two parameters constant and varying the third one, out of the $N \rightarrow$ Size of the Dimension, $D \rightarrow$ Number of Dimensions and $K \rightarrow$ Number of maximal sums.

For $D = 1, K = 50$

N	128*128	256*256	512*512	1024*1024
Speedup	10.123	11.124	12.042	13.123

On increasing the dimension size the speed-up increased this may be because in sequential there is an extra N−1 elements are added in the complete binary heap formation and these are again removed at final stage, whereas in parallel this extra load is not present.

For $N = 1024 * 1024, D = 1$

K	10	30	50	75	100
Speedup	13.223	13.124	13.123	12.931	12.823

On increasing K, the sequential algorithm has an extra load in the terms of clan sizes of the Frederickson Heap Selection Algorithm, whereas in parallel the number of subsections, i.e number of threads which explore there subsections increase. This increase in the K, leads to more wrap divergence and more subsections to explore, so a slight decrease in speedup with increase in K is expected.

For $N = 512 * 512, K = 100$

D	1	2	3
Speedup	11.823	12.524	13.242

On increasing the D value, the problem size increases drastically and parallel version accumulates the speedup at each 1-Dimensional array calculation so we have the speedup increasing with increasing the dimension value. Also for sequential version the extra nodes which are added and removed to form the complete binary heap increases in large scale. The speedup plots are shown in Fig.4 and Fig.5

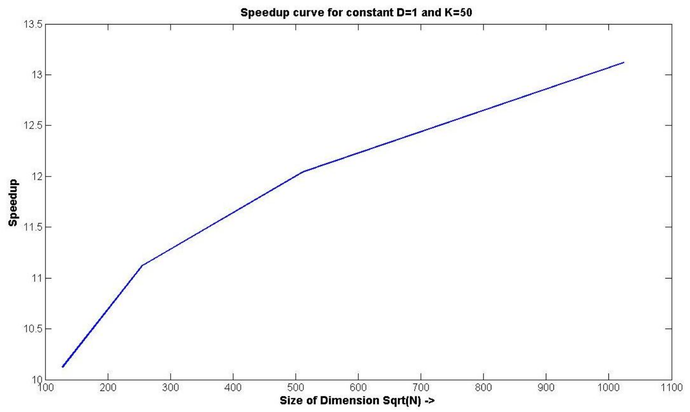


Fig. 4. Speedup Curve for Constant D and K

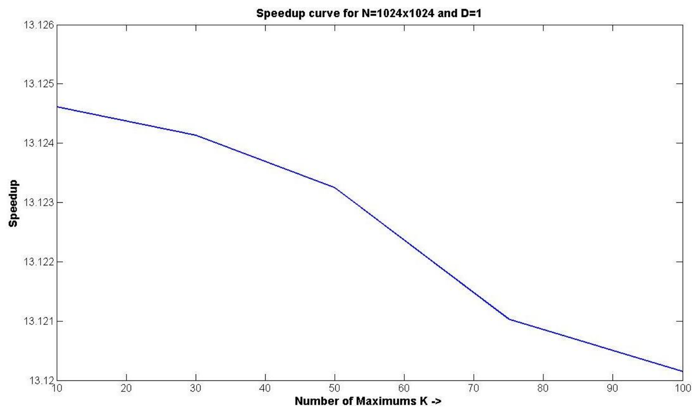


Fig. 5. Speedup Curve for Constant N and D

V. Conclusions and Future Work

A. Conclusions

Implemented the CUDA-based parallel version of the generalized D-dimensional K-Maximum Subarray Problem, parallelized most of the steps of the linear sequential algorithm. Instead of the high unbalanced load and highly sequential version, achieved a convenient speedup of upto 13X for a problem size of N=512*512, D=3 and K=100. And also we can conclude that even a lower bounded highly optimized sequential algorithm can have better results in parallel version.

B. Future Work

Due to time constraints as this is course project, the generalization for D-dimensional problem has not been completely optimized in all aspects and this can be handled in the future work. i.e, parallelizing some of the 1-D arrays calculations based on the N value. Writing different kernels based on different problem sizes. A comparison between the GPU implementation and a multi-threaded CPU implementation (Open-MPI and MPI versions) can be also considered in the future.

References

[1] S. E. Bae, "Sequential and parallel algorithms for the generalized maximum subarray problem," Ph.D. dissertation, National Taiwan University, 2007.

[2] G. S. Brodal and A. G. Jørgensen, "A linear time algorithm for the k maximal sums problem," in *Mathematical Foundations of Computer Science 2007*. Springer, 2007, pp. 442–453.

[3] S. Saleh, M. Abdellah, A. A. A. Raouf, and Y. M. Kadah, "High performance cuda-based implementation for the 2d version of the maximum subarray problem (msp)," in *Biomedical Engineering Conference (CIBEC), 2012 Cairo International*. IEEE, 2012, pp. 60–63.

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of computer and system sciences*, vol. 38, no. 1, pp. 86–124, 1989.

[5] G. N. Frederickson, "An optimal algorithm for selection in a min-heap," *Information and Computation*, vol. 104, pp. 197–197, 1993.