GPGPU-accelerated Interesting Interval Discovery and other Computations on GeoSpatial Datasets – A Summary of Results [*] [†]

Sushil K. Prasad
Georgia State University
Atlanta, GA
sprasad@gsu.edu

Shashi Shekhar
University of Minnesota
Minneapolis, MN
shekhar@cs.umn.edu

Michael McDermott
Georgia State University
Atlanta, GA
mmcdermott2@student.gsu.edu

Xun Zhou
University of Minnesota
Minneapolis, MN
xun@cs.umn.edu

Michael Evans
University of Minnesota
Minneapolis, MN
mevans@cs.umn.edu

Satish Puri
Georgia State University
Atlanta, GA
spuri@student.gsu.edu

ABSTRACT

It is imperative that for scalable solutions of GIS computations the modern hybrid architecture comprising a CPU-GPU pair is exploited fully. The existing parallel algorithms and data structures port reasonably well to multi-core CPUs, but poorly to GPGPUs because of latter's atypical fine-grained, single-instruction multiple-thread (SIMT) architecture, extreme memory hierarchy and coalesced access requirements, and delicate CPU-GPU coordination. Recently, our parallelization of the state-of-art interesting sequence discovery algorithms calculates one-dimensional interesting intervals over an image representing the normalized difference vegetation indices of Africa within 31 ms on an nVidia 480GTX. To our knowledge, this paper reports the first parallelization of these algorithms. This allowed us to process 612 images representing biweekly data from July 1981 through Dec 2006 within 22 seconds. We were also able to pipe the output to a display in almost real-time, which would interest climate scientists. We have also undertaken parallelization of two key tree-based data structures, namely R-tree and heap, and have employed parallel R-tree in polygon overlay system. These data structure parallelization are hard because of the underlying tree topology and the fine-grained computation leading to frequent access to such data structures severely stifling parallel efficiency.

[*]This invited paper extends our position paper in The NSF CyberGIS Project All-Hands Meeting, Seattle, 2013 [24]

[†]This material is based upon work supported by the National Science Foundation under Grant No. CCF 1048200, CNS 1205650 (Dr. Prasad), 1029711, IIS-1320580, 0940818 and IIS-1218168 as well as USDOD under Grant No. HM1582-08-1-0017, and HM0210-13-1-0005 (Dr. Shekhar).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
BIGSPATIAL '13:, November 05 - 08 2013, Orlando, FL, USA. Copyright 2013 ACM 978-1-4503-2534-9/13/11…$15.00.
http://dx.doi.org/10.1145/2534921.2535837

Keywords

Big Data, Analytics, Interesting Subpath, Ecological Change, CUDA-C, SIMT, R-Tree, Polygon Overlay

1. INTRODUCTION

1.1 Why GPGPU for GIS Data and Computation?

The computer architecture now is massively parallel and hybrid, with a pair of multi-core CPU and many-core GPGPU (with dozens of cores on a CPU and hundreds of cores on a GPGPU) now commonplace in laptops and desktops and the computer nodes of high-performance machines and clouds. Parallel and distributed processing is, therefore, imperative for data-and-compute-intensive geospatial computations, such as for polygonal overlay [30, 4, 5, 13, 29], evacuation routing [10, 7], and interesting interval/region discovery [33, 14, 15, 16] (some of which turn out to be "Big Data" problems). An efficient utilization of the CPU-GPGPU pair is critical, else the GIS programs will remain inefficient, incurring loss of one to two orders of magnitude in speedup, specially where strong-scaling is needed. In addition, from GIS user perspective, a key niche for GPGPU (in contrast with cloud computing or cluster) may lie in price-performance trade-off - most laptops and workstations come with GPUs providing opportunities for an order of magnitude faster response time for many GIS problems without additional hardware cost. The existing parallel algorithms and data structures port reasonably well to multi-core CPUs, but poorly to GPGPUs. Therefore, we foresee the need for significant undertaking by the geospatial and parallel processing communities to redesign the traditional parallel data structures and algorithms and discover new parallel techniques and tools, in general, and for GPGPU platform, in particular.

1.2 Interesting Interval/Region Discovery Problem

Discovering interesting intervals/regions from spatiotemporal (ST) datasets is important for applications such as understanding climate change, environmental monitoring, and public health. In climatology, subsequence analysis of time series data allows for the prediction of weather patterns and to study and build predictive ecological models. In GIS,

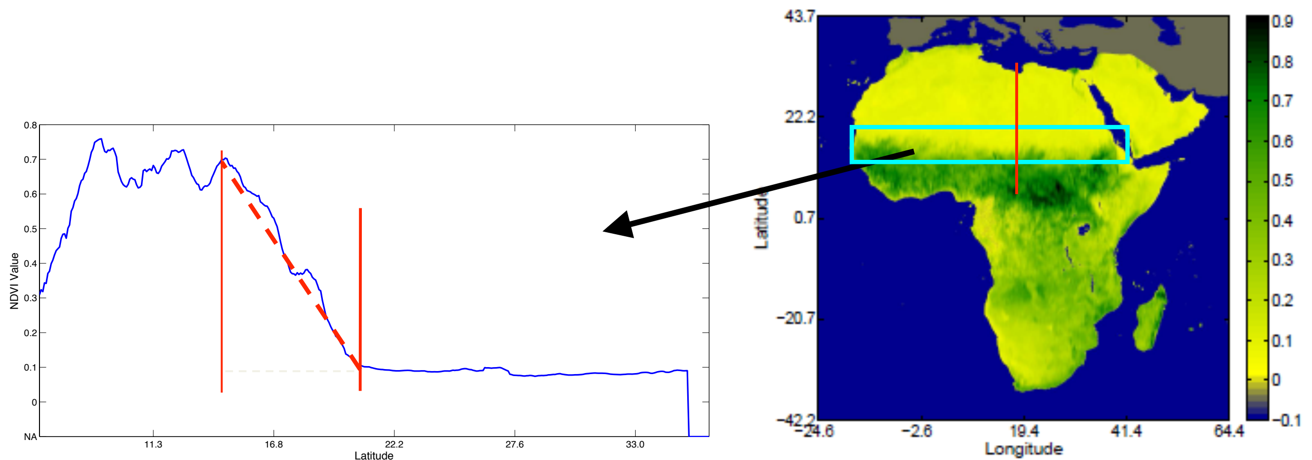

Figure 1: Sahel Region Abrupt Change [33]

subsequence analysis of time series data allows for identifying spatiotemporal regions of interest such as in tracking and predicting migratory patterns, correlating ecological phenomena, tracking the spread of disease, and search and rescue.

Subsequence analysis involves searching through time series data for groups of data points that pass a relevant interest threshold. A typical naive approach exhaustively examines every possible subsequence. This type of exhaustive search is a very time consuming procedure. On small datasets computation time can be relatively fast, however, on large datasets such as in average climatology or bioinformatics knowledge discovery, computations times very quickly become very large. It is therefore very important to find ways of speeding up these searches.

We consider the problem of finding areas of abrupt change of vegetation, which may allow us to detect patterns of desertification and deforestation over time. For the purpose of this paper, our dataset is 26 years of bi-weekly normalized difference vegetative index (NDVI) data for Africa starting in 1981 [9]. The problem becomes finding longitudinal sequences in NDVI data that increase or decrease faster than a certain threshold. Formally, *"Given a spatiotemporal path S consisting of n locations* S_1*,* S_2*, ...* S_n*, a sub-path* $W =$ *(i, j) of S is a contiguous subset of locations from* S_i *to* S_j *in S. An interesting subpath is such a path which satisfies a given interest measure such as in equation 1 [33]."* Figure 1 indicates one part of a longitudinal band being analyzed in red on the right and its graph on the left. The Sahel region of Africa is surrounded by a cyan box on the right, and in the graph it is represented by the enclosing vertical red bars.

The dataset is searched to find all the longest continuous longitudinal bands of sharp increase in NDVI value, which indicates abrupt increase in vegetation for the band. For instance, in Figure 1, we can see the abrupt decrease in vegetation across the Sahel region as it transitions from savannah to desert. When studied as time series this shows the areas of rapid change in vegetation as seen in desertification and deforestation and we can computationally study and correlate the growth/shrinkage of the desert areas with other spatio-temporal and geolocated environmental factors.

Subsequence discovery on the NDVI geotiff data [9] lends itself well to the GPU/CUDA environment. Leveraging the rectangular matrix structure of the data, the relatively low branching of the subsequence interest measure, and the floating point compute capabilities of CUDA, we are able to reduce the execution time by an order of magnitude over the comparable sequential algorithm. This paper describes our GPU-based design and implementation of an algorithm for detection of 1-D interesting intervals based on the state-of-art algorithms described in [33]. In addition to significant speedups obtained, this case-study can provide a road-map for GIS scientists for parallelizing similar workloads.

We conclude with summarizing our work on parallelizing R-tree and heap data structures, and developing a polygonal overlay system employing R-trees.

2. RELATED WORK

Early work on interesting interval discovery only aim to find interesting points in a time series for abrupt change detection [23, 28]. [14] proposed a SWEET approach to discover anomalous temporal intervals in a pair of sensor reading series measured from a river. [33] formulated the interesting sub-path discovery problem as finding contiguous subsets of a given ST path (e.g., a longitudinal paths or a time series) that are interesting according to a user-defined interest measure function. A Sub-path Enumeration and Pruning (SEP) approach was proposed to find ST intervals with rapid changes, such as ecotones (e.g., the Sahel zone) and periods of climate changes (e.g., rainfall decrease).

Other techniques discover interesting spatial regions (2D) or space-time volumes (3D) in ST datasets. The spatial scan statistic [15] and its variation, ST scan statistics [16], have been designed to detection regions (and time periods) of disease outbreaks. A recent work by Zhou et al. [34] investigated the problem of finding persistent change windows (region and time interval) in earth science datasets.

To our knowledge, this paper reports first such parallelization of interesting interval discovery algorithms.

3. ALGORITHM DESCRIPTION AND METHODOLOGY

3.1 Sequential Algorithms

The naive algorithms in both their parallel and sequential form are brute force algorithms which suffer from inefficiency due to performing many redundant operations and in the case of the parallel algorithm breaking many of the paradigms necessary for efficient computation on the GPU (see Section 3.3).

The sequential row-wise SEP algorithm [33] which significantly improves upon the naive algorithm has three parts: a scan phase wherein a lookup table is built, a discovery phase wherein all subsequences are found and an elimination phase where all dominated subsequences are removed.

1. The scan phase starts by first scanning all unit-size subpaths in a longitudinal band to build a lookup table of values [33]. This lookup table functions much like a prefix sums. This is $O(n)$ for a single longitudinal band and $O(n^2)$ for the whole NDVI image.

2. The discovery phase finds all longest subsequences in a longitudinal path by starting at the longest potential subsequence and computing its interest measure, then doing the same for the next longest, and so on until it finds an interesting sub-path. Our interest measure is defined as follows which should exceed a minimum score/threshold:

$$Interest\ Measure = \frac{Average\ value\ of\ the\ unit\ subpath > threshold}{Average\ value\ of\ the\ all\ unit\ subpaths} \quad (1)$$

This computation is what makes the lookup table from the scan phase so effective. It allows us to not do a large portion of these calculations by simply looking at what we previously did and adding new information to it. This is $O(n^2)$ comparisons for a single longitudinal band and $O(n^3)$ for the entire NDVI image in the worst-case.

3. The elimination phase takes all the subsequences returned by the discovery phase and drops those subsequences that are dominated by (subset of) larger subsequences. It preserves subsequences where only one end is being overlapped. It does this by comparing the endpoints of each subsequence and seeing if those points fall inside the bounds of another subsequence. This phase guarantees the correctness of the algorithm. This is also $O(n^2)$ for a single longitudinal band and $O(n^3)$ for the entire NDVI image.

Therefore, the entire SEP algorithm is $O(n^2)$ for a single longitudinal band and $O(n^3)$ for the entire image. This is because we have to do all three steps for each longitudinal band of the image and there are n longitudinal bands to consider.

3.2 Parallel Algorithms and Implementations

The GPU implementations closely mirror the sequential algorithms of [33] - the major difference is in the discovery phase. Instead of iterating through every item in a longitudinal band we introduce massively parallel allocation of the workload where a thread processes only a minimal number of items. The preprocessing steps are largely identical. The Parallel row-wise SEP has an additional pre-processing step and an additional post-processing step that transpose and reverse transpose the data, respectively, in order to coalesce global memory accesses.

Algorithm 1 Parallel Naive

1: **for** i n **to** *threadID* **do in parallel** ▷ discovery
2: *interest* *comput interest measure*
3: **if** *interest* $>$ *threshold* **then**
4: *add to candidate list*
5: **end if**
6: **end parallel for**
7: A *sequence starting at threadID*
8: **for** i *threadID* **to** $A.end$ **do in parallel**
9: ▷ elimination
10: $B \leftarrow$ *sequence starting at* $threadID + i$
11: **if** *A dominates B* **then**
12: *drop B from candidates*
13: **end if**
14: **end parallel for**
15: **return** *candidates*

The naive parallel implementation does not use a lookup table and relies on mere parallelism in order to realize a speedup. This means that for every interest measure calculation in a longitudinal band it must fetch large quantities of single data items from the global memory of the GPU and then perform large numbers of calculations on line 3. It must do this type of inefficient fetch from global memory again in the elimination phase and, because each elimination step is of no consistent length, suffers from large amounts of thread divergence (see Section 3.3).

Algorithm 2 Parallel Row-Wise SEP)

1: **for** i n **to** *threadID* **do in parallel** ▷ lookup and discovery
2: *compute lookup table entry for threadID*
3: *interest* $\leftarrow$ *compute interest measure*
4: **if** interest > threshold **then**
5: *add to candidate list*
6: **end if**
7: **end parallel for**
8: *bounds* $\leftarrow$ *largest candidate in block*
9: $A \leftarrow$ *sequence starting at threadID*
10: **for** i $(A.start - bounds)$ **to** $(A.end + bounds)$ **do in parallel** ▷ elimination
11: $B \leftarrow$ *sequence starting at* i
12: **if** *B dominates A* **then**
13: *drop A from candidates*
14: **end if**
15: **if** *A dominates B* **then**
16: *drop B from candidates*
17: **end if**
18: **end parallel for**
19: **return** *candidates*

In contrast, the parallel row-wise SEP algorithm constructs a lookup table in $O(n/p)$ time per longitudinal band, where

n is the size of a column and p is the number of processors/cores. It must do this for each band rending this in effect $O(n^2/p)$. It then uses this lookup table in the discovery phase on line 4 which eliminates the needlessly redundant computation that the naive algorithm does on line 3. It also performs its fetches from global memory in a coalesced fashion. In the elimination phase it avoids large amounts of thread divergence by utilizing shared memory to ensure that each thread in a warp performs almost the same number of calculations (see Section 3.3). This elimination step, because we limit the number of calculations it does, is $O(m/p)$ where m is the length of the largest candidate subsequence. Because we do this for each longitudinal column and because the largest possible candidate subsequence is of length n, this becomes $O(n^2/p)$ as well. Finally, because we launch n/2 threads for each longitudinal column in all three phases, the overall computational complexity can become $O(n)$.

3.3 Extracting Parallel Performance using CUDA: General Guidelines

Our approach employs the GPU to do these searches in a massively parallel way. This requires some important paradigms to be incorporated into the implementation which seek to maximize the strengths of the GPU while reducing its weaknesses and are required to realize the best speedup in computation time.

The first of these paradigms is how you tell the GPU how many threads you want to launch. CUDA breaks this into two parts: threads per block and blocks per grid [11]. A block is a group of threads that can all read and write to the same shared memory and can be coordinated to some degree. There is no coordination between blocks other than through reading and writing to global memory. A grid is simply a group of blocks. There are two main types of memory that each thread has access to. There is shared memory which is very limited (48 kilobytes in our case) but is many times faster than global memory which is much larger (1 gigabyte or more). In general you want to use the maximum appropriate number of threads for a given task, but (as we found out) this may not always be ideal. Instead of launching a thread for every possible subsequence naively, we launch one thread for every two possible subsequences.

The second of these paradigms is to minimize the communication between the CPU and the GPU. This is particularly hard in the case of certain bioinformatics problems due to the large memory footprint of the dataset compared to size of the memory available on the GPU. In the case of our experimental data, there was adequate memory on the GPU to store the complete search space needed (i.e., an NDVI image). This allows us to read at least a single image at a time into the GPU's global memory which ultimately allows us to generate an almost real-time visualization as the processing is done.

The third paradigm is to maximize occupancy of the processors. Occupancy is the ratio of active warps to total warps in a processor, a warp being a group of 32 threads. If occupancy is not as close to 100% as possible then the GPU is not working as efficiently as it can. It should be noted that achieving higher occupancy is not a guarantee of higher performance and 100% occupancy is impractical/impossible. This has to do with how the GPU schedules threads in order to hide the latency associated with warps reading from memory, writing to memory, or waiting on other warps [20]. The general rule is that you want to have the maximum number of threads running on a given GPU processor. The easiest way to do this is to utilize the largest blocks of threads possible. Our implementation uses 576 threads and 1072 blocks. This increased the workload of each thread by a factor of 2 over the naive implementation and allowed the warps to contain a full 32 threads each.

The fourth paradigm is to minimize accesses to global memory. In the naive parallel algorithm, we did not minimize access to global memory or minimize branching. In the non-naive version, global memory accesses were kept to a minimum. However, due to the size of the data being analyzed, it was still necessary to make many reads and writes. In cases like this, it is best to do this in a coalesced fashion, which is to read from memory in such a way as to allow the threads in a half-warp to access global memory from a contiguous part of global memory at once [11]. So $thread_0$ reads $memory_0$, $thread_1$ reads $memory_1$, ... , $thread_n$ reads $memory_n$. In our implementation, each block reads an entire longitudinal column at once into shared memory, computations are done using that shared memory and then each block writes that entire new longitudinal column back out to global memory.

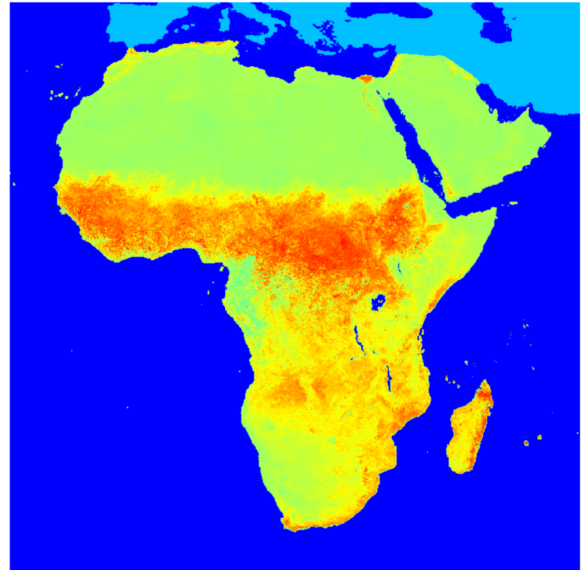

Figure 2: False Color Africa NDVI Image [9]

The final paradigm is to minimize branch divergence and to keep all your threads in a block doing the same thing when possible. Branch divergence in CUDA must be kept to a minimum simply because it has the effect of reducing parallelism [21]. Very simplistically, for a warp of threads if the first half of the warp is executing true part of an if-else statement and the other half is not, the GPU will 'pause' the second half while the first half executes then pause the first half while the second half executes in order to keep all the threads in a warp working on the same instruction. This can happen each time we have a branch in our code. Enough branching like this could cause a GPU algorithm implementation to run much slower than its sequential ver-

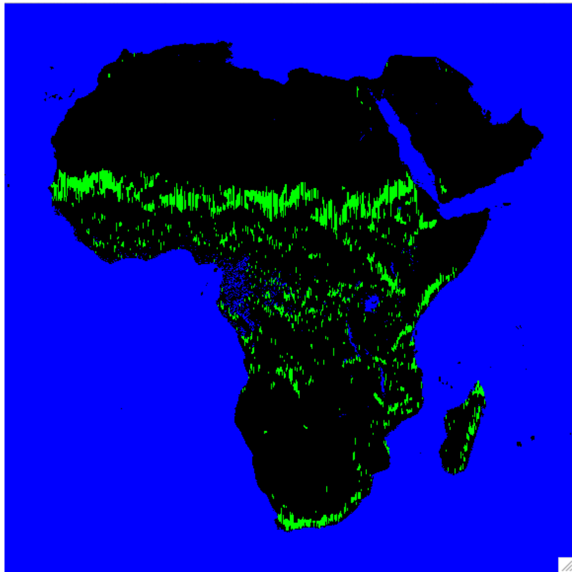

Figure 3: Interesting Subpaths Highlighted

sion and in fact this type of behavior is seen somewhat in the naive parallel implementation [11]. The rule of thumb here is that you always want all your threads in a given warp doing the same thing at all times. In the elimination step, this was leveraged by making sure each thread performed the same number of comparisons. The naive version of this step does significantly fewer comparisons in contrast but is overall much slower because of branch divergence.

3.4 Major Optimizations

As previously mentioned, the parallel naive implementation does not utilize all of the strengths of CUDA architecture. For an 1152×1152 image, this means that the naive implementation launches a thread for every item in the matrix, roughly 1.3 million threads. The row-wise algorithm only needs half as many threads to outperform the naive algorithm due to the use of the lookup table, optimized memory access and minimized thread divergence in the elimination step.

The optimized memory access is achieved by first transposing the matrix in a coalesced fashion [11], then reading a whole row (corresponding to a full longitudinal column) into the shared memory from the texture memory (which is a type of spatially-arranged read-only global memory [22]) for a given thread block. Computations can then be done by a thread block by reading/writing to its shared memory and output is then written back out to global memory in a coalesced fashion [11]. This step increases memory bandwidth utilization by ensuring that when we need to read or write to global memory we are doing it in large contiguous blocks defined by our warps instead of by threads within a warp in a non-aligned fashion. We discussed this earlier with thread divergence within a warp. Thread divergence coupled with data-independent global memory reads and writes throttle the performance due to the very slow nature of global memory. The difference between the timing for the naive parallel and the non-naive SEP parallel implementations illustrate this problem. Even though the non-naive implementation runs with half as many threads and actually does more computation, it runs many times faster than the naive implementation.

When high fidelity in output correctness is needed the elimination phase can be run *explicitly*. This adds to the execution time but guarantees that all the dominated subsequences are removed and all intersecting subsequences which are not dominated are preserved. For the purposes of visualization of the interest measure, this may not be needed and the elimination is done *implicitly* by simply allowing the visualization to render all subsequences including the dominated subsequences which would just be 'overwritten' by the dominating subsequences.

Figure 2 is a false color representation of the NDVI input data for Africa. Figure 3 is the output results that have been processed to show areas of abrupt increase in green. Black areas represent areas of change below the threshold. Blue and Cyan areas indicate water or areas for which there is no data.

4. EXPERIMENTAL RESULTS

The results of the GPU implementation are promising (Figure 4). Almost real-time visualizations are possible when running on an older consumer-grade nVidia GTX 480 using an openGL render loop [26]. The elimination step is done implicitly instead of explicitly in this case. The jump in speedup between the naive and row-wise versions of the implementation is significant. The parallel naive version runs in 2701ms while the parallel row-wise version runs in 56 ms (30.65 ms with implicit elimination) when processing a single NDVI geotiff. This is almost a 50 times speedup achieved through the use of a lookup table and basic memory optimizations. When processing the entire NDVI dataset for Africa consisting of 612 1152×1152 geotiffs, this difference quickly becomes very noticeable. The sequential processing time for the visualization of the whole dataset for Afica is 9.6 minutes; when running the naive parallel algorithm with implicit elimination this time is significantly longer at 26 to 34 minutes. The parallel row-wise SEP algorithm, however, reduces this time to an acceptable 20 to 26 seconds when running the row-wise algorithm with implicit elimination; with explicit elimination this is 21 to 27 seconds. This speedup allows for near real-time visualization.

5. CURRENT WORK

Currently, work is being done on a Hadoop implementation of the algorithm in conjunction with GPU-parallelization. This presents several challenges, one of which is interfacing CUDA-C Kernel code with the Java implementation of Apache Hadoop. A second challenge is that the individual images are rather small to work with in Hadoop which causes what is known as "the small file problem" to occur [31].

On our other GPGPU work on GIS datasets, we have undertaken parallelization of two key tree-based data structures, namely R-tree and heap, and have employed parallel R-tree in a polygon overlay system. These data structure parallelization are hard because of the underlying tree topology and the fine-grained computation leading to frequent access to such data structures severely stifling parallel efficiency. Therefore, the current best parallelization of R-tree on GPU

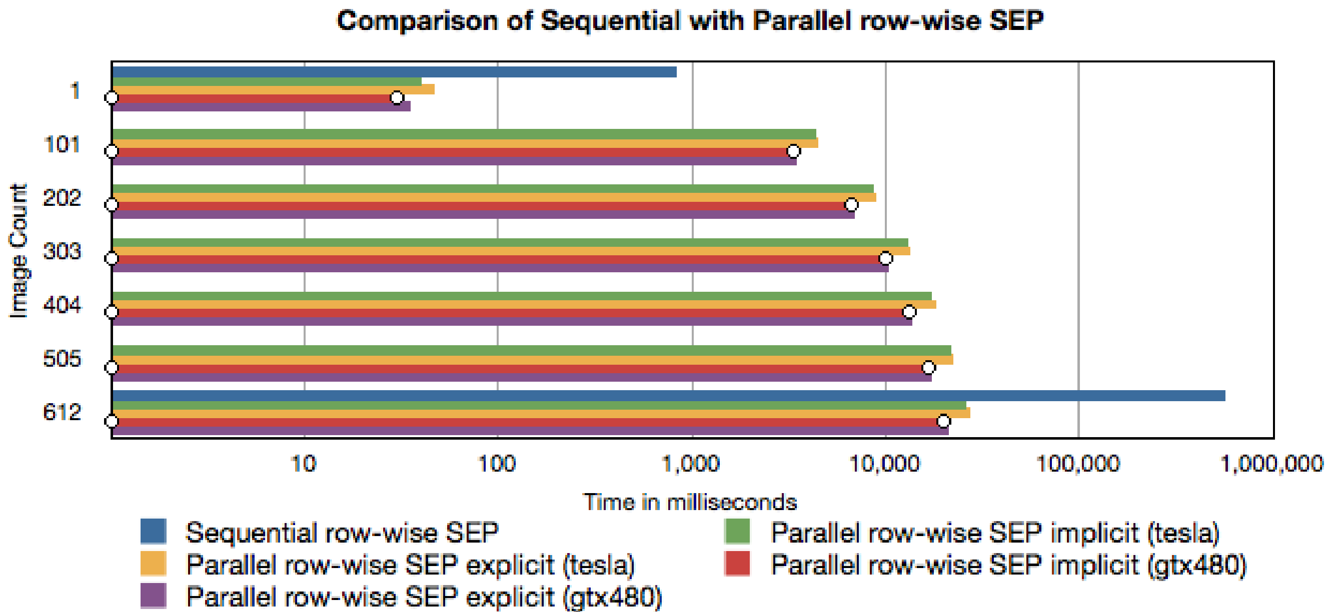

Figure 4: Timings are representative of only the lookup, discovery, and elimination phases

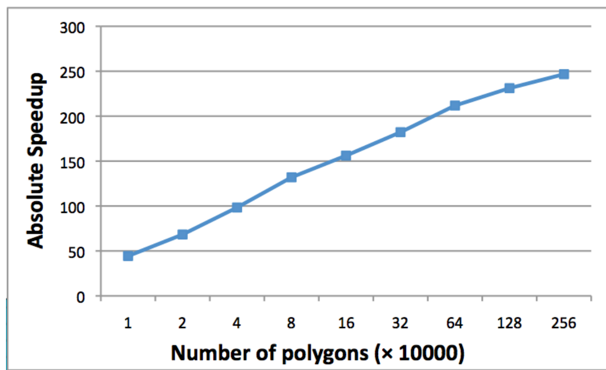

Figure 5: R-Tree Construction

was limited to about 20-fold speedup [17, 19, 6, 27]. We summarize our current results as follows.

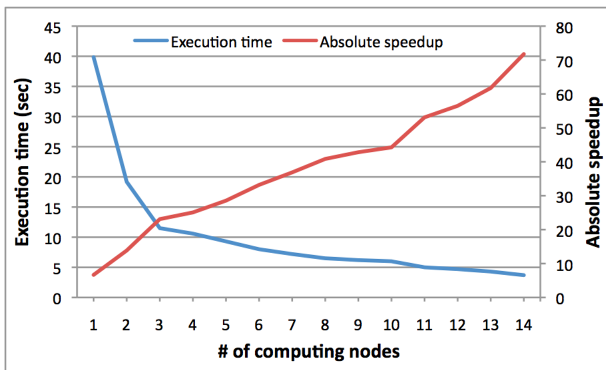

Figure 6: Polygonal Overlay System

Parallel R-Tree: Our new parallel algorithms for construction and search has yielded the first demonstrated 200-fold speedup in R-tree construction on a GPU (patent pending, [8]). This would be useful for large-scale range querying and can serve as template for parallelizing other tree-based structures such as MVR tree, TPR tree, and other generalizations of spatio-temporal datasets.

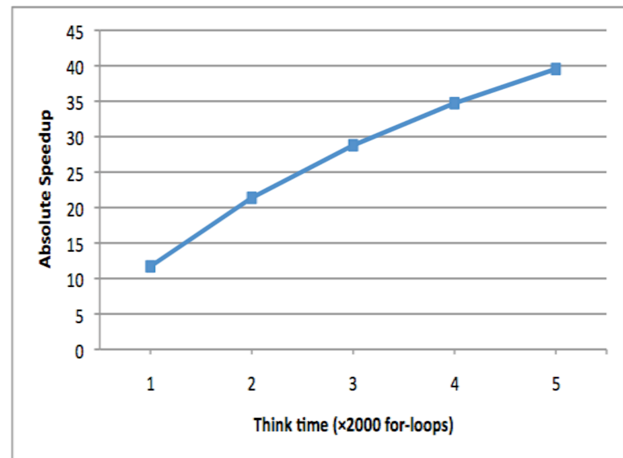

Figure 7: Parallel Priority Queue

Parallel Polygon Overlay System: Our GIS system employing the parallel R-tree can process about 200K polygons within a few seconds with 70-fold speedup on a (12-node Linux cluster that previously took tens of minutes [25, 12, 1, 3, 2]). Thus, it has the potential for bringing a practical overlay tool to the Geo Scientists.

Parallel Priority Queue: Our parallel heap data structure supports large batches of extracting highest priority items and inserting newly produced items with 30-fold speedups

(patent pending, [12]). This has potential application to shortest path and evacuation route planning. For latter, the current algorithms are sequential and slow. For instance, state of art CCRP routing algorithm took more than a day of computation time to compute evacuation routes and schedules for San Francisco [18, 32].

6. REFERENCES

[1] D. Agarwal and S. Prasad. Lessons Learnt from the Development of GIS Application on Azure Cloud Platform. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 352–359, 2012.

[2] D. Agarwal, S. Puri, X. He, , and S. K. Prasad. Cloud Computing for Fundamental Spatial Operations on Polygonal GIS Data. In Cloud Futures 2012 - Hot Topics in research and education, may 2012.

[3] D. Agarwal, S. Puri, X. He, and S. Prasad. A System for GIS Polygonal Overlay Computation on Linux Cluster – An Experience and Performance Report. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1433–1439, 2012.

[4] T. M. Chan. A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections. In *In Proc. 6th Canad. Conf. Comput. Geom*, pages 263–268, 1994.

[5] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 590–600, 1988.

[6] J. K. Chen, Y. F. Huang, and Y. H. Chin. A study of Concurrent Operations on R-trees. *Inf. Sci.*, 98(1-4):263–300, May 1997.

[7] T. J. Cova and J. P. Johnson. A Network Flow Model for Lane-based Evacuation Routing. *TRANSPORTATION RESEARCH PART A*, 37:579–604, 2003.

[8] S. P. et al. GPU-based Parallel R-tree Construction and Querying and Its Application to GIS Polygon Overlay Processing, *Provisional Patent Application*, 2013.

[9] Global Land Cover Facility. Ndvi dataset. `http://glcf.umd.edu/data/ndvi`.

[10] H. Hamacher and S. Tjandra. In *Pedestrian and Evacuation Dynamics, vol. 2002, pp. 227–266,*, 2002.

[11] M. Harris. An Efficient Matrix Transpose in CUDA C/C++. `https://developer.nvidia.com/content/efficient-matrix-transpose-cuda-cc`, Feb. 2013.

[12] X. He, D. Agarwal, and S. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, 2012.

[13] R. G. Healey, M. J. Minetar, and S. Dowers, editors. *Parallel Processing Algorithms for GIS*. Taylor & Francis, Inc., Bristol, PA, USA, 1997.

[14] J. Kang, S. Shekhar, C. Wennen, and P. Novak. Discovering Flow Anomalies: A SWEET Approach. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, pages 851–856, 2008.

[15] M. Kulldorff. A spatial scan statistic. *Communications in Statistics-Theory and Methods*, 26(6):1481–1496, 1997.

[16] M. Kulldorff, R. Heffernan, J. Hartman, R. Assunção, and F. Mostashari. A Space-Time Permutation Scan Statistic for Disease Outbreak Detection. *PLoS Med*, 2(3):e59, 02 2005.

[17] M. Kunjir and A. Manthramurthy. Using Graphics Processing in Spatial Indexing Algorithms. In *Research report, Indian Institute of Science, Database Systems Lab*, 2009.

[18] Q. Lu, B. George, and S. Shekhar. Capacity Constrained Routing Algorithms for Evacuation Planning: A Summary of Results. In *Proceedings of the 9th international conference on Advances in Spatial and Temporal Databases*, SSTD'05, pages 291–307, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] L. Luo, M. D. F. Wong, and L. Leong. Parallel implementation of R-trees on the GPU. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 353–358, 2012.

[20] nVidia. Cuda C Best Practices Guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#execution-configuration-optimizations`, July 2013.

[21] nVidia. Cuda C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, July 2013.

[22] nVidia. Cuda Toolkit Documentation. `http://docs.nvidia.com/cuda/index.html`, Aug. 2013.

[23] E. S. PAGE. Continuous Inspection Schemes. *Biometrika*, 41(1-2):100–115, 1954.

[24] S. Prasad, S. Shekhar, X. He, S. Puri, M. McDermott, X. Zhou, and M. Evans. GPGPU-based Data Structures and Algorithms for Geospatial Computation - A Summary of Results and Future Roadmap. In *Position Paper at The All Hands Meeting of the NSF CyberGIS Project*, Sept. 2013.

[25] S. Puri, D. Agarwal, X. He, and S. K. Prasad. MapReduce algorithms for GIS Polygonal Overlay Processing. In *IEEE International Parallel and Distributed Processing Symposium workshops (HPGC)*, 2013.

[26] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[27] B. Schnitzer and S. Leutenegger. Master-client R-trees: a new parallel R-tree architecture. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, pages 68–77, 1999.

[28] M. Sharifzadeh, F. Azmoodeh, and C. Shahabi. Change detection in time series data using wavelet footprints. In *Proceedings of the 9th international conference on Advances in Spatial and Temporal Databases*, SSTD'05, pages 127–144, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] F. Wang. A Parallel Intersection Algorithm for Vector Polygon Overlay. *IEEE Comput. Graph. Appl.*, 13(2):74–81, Mar. 1993.

[30] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. *Proc.*

VLDB Endow., 5(11):1543–1554, July 2012.

[31] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, may 2012.

[32] K. Yang, V. Gunturi, and S. Shekhar. A Dartboard Network Cut Based Approach to Evacuation Route Planning: A Summary of Results. In N. Xiao, M.-P. Kwan, M. Goodchild, and S. Shekhar, editors, *Geographic Information Science*, volume 7478 of *Lecture Notes in Computer Science*, pages 325–339. Springer Berlin Heidelberg, 2012.

[33] X. Zhou, S. Shekhar, P. Mohan, S. Liess, and P. K. Snyder. Discovering interesting sub-paths in spatiotemporal datasets: a summary of results. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 44–53, New York, NY, USA, 2011. ACM.

[34] X. Zhou, S. Shekhar, and D. Oliver. Discovering Persistent Change Windows in Spatiotemporal Datasets: A Summary of Results. In *In Proc. 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial-2013)*, November 5 2013 (to appear).