Accelerating Spatial Autocorrelation Computation with Parallelization, Vectorization and Memory Access Optimization

With a focus on rapid recalculation of COVID related spatial statistics for faster geospatial analysis and response

Anmol Paudel
Department of Computer Science
Marquette University
Milwaukee, USA
anmol.paudel@marquette.edu

Satish Puri
Department of Computer Science
Marquette University
Milwaukee, USA
satish.puri@marquette.edu

***Abstract*—Geographic information systems deal with spatial data and its analysis. Spatial data contains many attributes with location information. Spatial autocorrelation is a fundamental concept in spatial analysis. It suggests that similar objects tend to cluster in geographic space. Hotspots, an example of autocorrelation, are statistically significant clusters of spatial data. Other autocorrelation measures like Moran's I are used to quantify spatial dependence.**

Large scale spatial autocorrelation methods are compute-intensive. Fast methods for hotspots detection and analysis are crucial in recent times of COVID-19 pandemic. Therefore, we have developed parallelization methods on heterogeneous CPU and GPU environments. To the best of our knowledge, this is the first GPU and SIMD-based design and implementation of autocorrelation kernels. Earlier methods in literature introduced cluster-based and MapReduce-based parallelization. We have used Intrinsics to exploit SIMD parallelism on x86 CPU architecture. We have used MPI Graph Topology to minimize inter-process communication.

Our benchmarks for CPU/GPU optimizations gain upto 750X relative speedup with a 8 GPU setup when compared to baseline sequential implementation. Compared to the best implementation using OpenMP + R-tree data structure on a single compute node, our accelerated hotspots benchmark gains a 25X speedup. For real world US counties and COVID data evolution calculated over 500 days, we gain upto 110X speedup reducing time from 33 minutes to 0.3 minutes.

***Index Terms*—Spatial Statistic, Getis-Ord, Moran's I, Geary's C, Hotspots, Vectorization, Cache, Intrinsics, Parallelization, OpenMP, OpenACC, CUDA, MPI, MPI Topology**

I. Introduction

In spatial statistics and spatial data mining, there are many methods to discover and explore interesting patterns in spatial data. Spatial autocorrelation is one such class of methods that are used in spatial data analysis. Spatial datasets often are not independent and identically distributed (i.i.d) [22]. Spatial datasets exhibit statistically significant clustering in attribute values under study.

Hotspots analysis is a technique in geospatial analysis used to visualize geographic data in order to show areas where a higher density or cluster of activity occurs. For example, in a city, we can collect crime data from different locations and with hotspot analysis we can see if there are clusters in the city with significantly higher/lower incidence of crime than so by random chance. Two concepts - similarity of values and proximity of locations, or lack of those, are crucial to calculating hotspots and hence requires spatial statistics. Hotspot detection is useful in many fields like public health, crime analysis, schooling, sales, agriculture etc.

We focus on Getis-Ord (Gi*) statistic which is computed for each feature in a dataset. The resultant z-scores and p-values show where features with either high (or low values) cluster spatially. In short, each feature is evaluated within the context of neighboring features. To be a statistically significant hotspot, a feature will have a high value and be surrounded by other features with high values as well.

Hotspots are sometimes confused with a similar spatial visualization technique known as heatmaps. Hotspots differ from heatmaps where point data is analyzed in order to create an interpolated surface showing the density of occurrence where each cell is assigned a density value and the entire layer is visualized using a gradient.

We present performance engineering for Hotspots kernel using SIMD on CPUs and SIMT (Single Instruction Multiple Thread) on GPUs for exploiting fine-grained vector/data parallelism. For relative speedup calculations, we have used sequential implementation with spatial sorting as a baseline. For absolute speedup calculation, we have used R-tree data structure based implementation. Based on this R-tree baseline, we have demonstrated absolute speedup upto 16X using SIMD + multi-threading on a single compute node. For scalability, our system leverages multiple GPUs using MPI. Our benchmarks for CPU/GPU optimizations gain upto 750X relative speedup with a 8 GPU setup when compared to baseline sequential implementation.

Earlier methods for hotspots problem have used pointer-based tree data structures like quadtree for storing location data and for range query. For effective SIMD/SIMT parallelization,

instead of tree data structure, we have designed a novel spatial locality-preserving 2D array-based data structure for weight matrix. On a distributed memory environment, this weight matrix further aids in creating task interaction graph which can be utilized to minimize communication using MPI graph topology functions.

The rest of the paper is organized as follows. Section II presents the motivation and background. Section III presents the parallel formulation for the problem. Section IV presents the acceleration techniques on CPUs and GPUs. Section V presents the experimental results. Finally, we conclude in Section VI.

II. Motivation and Background

Finding patterns helps us identify causes and predict future trends. For instance, finding hotspots of Covid-19 occurrences enable us to study disease spread and efficient resource allocation to combat the problem at hand. We have identified important autocorrelation kernels in spatial domains for parallelization. In the existing work, the focus has been on coarse-grained approaches with less attention to data movement aspects and communication complexity [23].

A. Spatial autocorrelation

The notion of spatial autocorrelation is related to first law of geography: Everything is related to everything else, but nearby things are more related than distant things [24]. The value of attributes at a given location tend to vary gradually over space. For instance, weather of two adjacent areas tend to be similar. In many cases, events in a given area are influenced by the events at neighboring areas. In spatial statistics, this property is called spatial autocorrelation [22]. A famous example of application of this concept was finding the link between Cholera outbreak and contaminated water in London in 1855 by looking at the clustering of disease occurrences (hotspots) around a water pump. An example of hotspots map is shown in Figure 1.

Spatial interdependence of attributes exhibited in data with respect to location and distance is captured by statistical measures like Moran's I. There are many local and global auto-correlation kernels. We focus on a representative and popular kernel - Hotspots. For a set of disease occurrences, finding hotspots aim at detecting disease outbreaks well before it results in a large number of cases. Hotspots are statistically significant clusters of observations based on similarities of values and locations. Hotspot detection is used in many fields like public health, crime analysis, etc.

B. Common Dataset Structures

Data for geo-spatial autocorelation analysis can usually come in 3 forms:

1) Aggregated Boundary data: This is the most typical type of available dataset for which usually a boundary is given and a value corresponding to the boundary is available. The boundary can be a known regular shape like square, rectangular, hexagonal or an irregular polygonal boundary. An example of this would be county level covid cases data. For each county, there is a defined polygonal boundary which is not a regular shape and for each county there would be a corresponding attribute value like active covid cases.
2) Unit point incidence data: This is the type of data where we have geolocation instances of incidents. Here we would have multiple points where each point corresponds to a single incident. Common example of this type of dataset is the crime dataset where each point relates to a reported criminal activity. A covid related example would be having a dataset of all the people who tested positive in a given area. In this dataset, each person would represent an individual incident and the geolocation of their home address would be an incident point.
3) Aggregated point incidence data: This is the type of data where we have instances from an area aggregated at a point. In the crime dataset, the geolocation of the police station could be the incident point and number of complaints are aggregated to get one single attribute value per incident point. A covid related example would

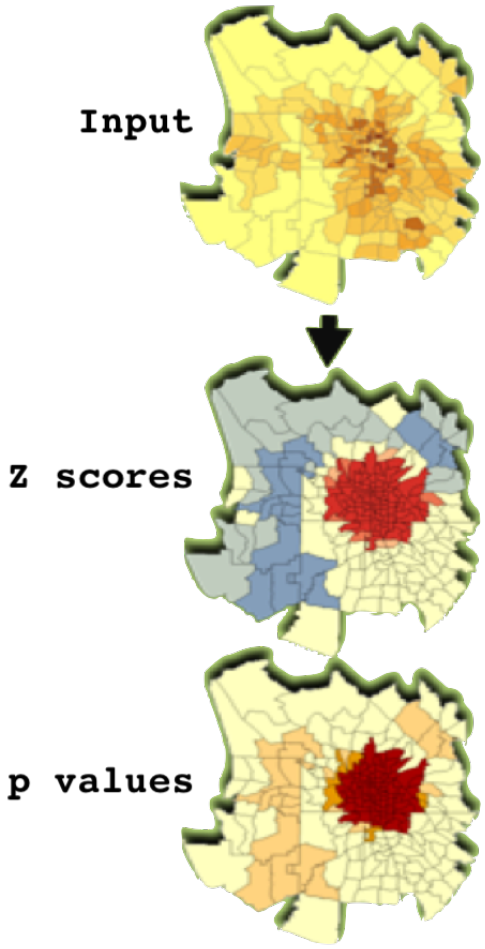

Fig. 1. Polygon boundaries with their corresponding z scores and p values [1]

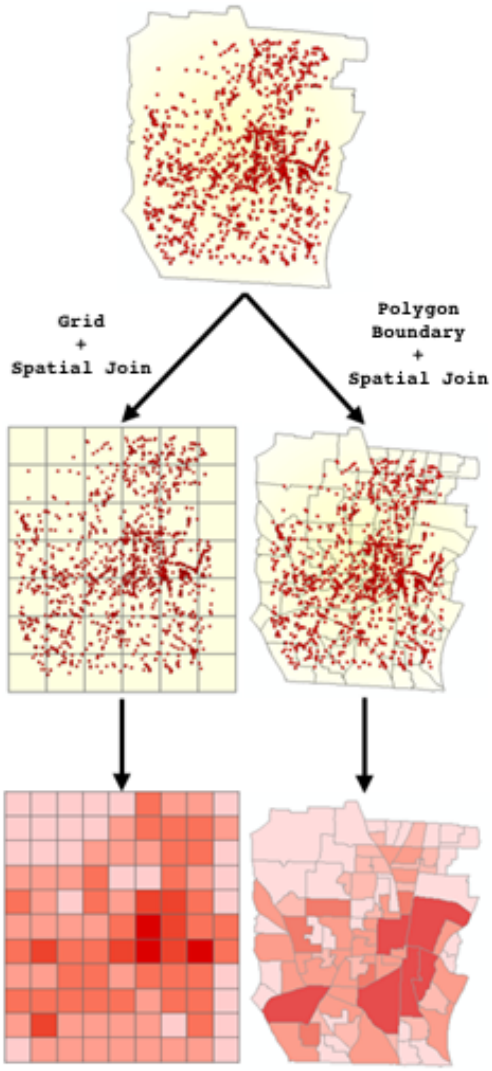

Fig. 2. Point data overlaid on a Grid vs Polygonal Boundaries [1].

be having a list of rapid testing centers, where the geolocation of the testing center is the incident point and the number of all tested positive cases are the aggregate attribute value.

In geospatial analysis, to calculate and show hotspots, boundaries are required. In the second case, the data can be overlaid on a regular grid of squares, rectangles, or hexagonal shapes. Another approach is to overlay the data on top of a polygonal layer, for instance, boundaries of zipcodes. All the values inside the boundary can be aggregated and used as the corresponding attribute value for the polygonal boundary. Figure 2 shows an example of data being overlaid on a regular grid and a polygonal map. Depending on the choice of data overlay, the computational cost will vary.

C. Parallelization

Vector/SIMD Intrinsics: Vector/SIMD extensions of Instruction Set Architecture are provided by modern CPUs for single instruction steam, multiple data stream (SIMD) processing. For x86 CPUs, special wide registers and vector instructions are provided for parallel processing at the instruction set level. For instance, x86 processors provide AVX (advanced vector extensions) instructions. ARM processors provide neon extensions. In this paper, for effective SIMD parallelization, we have used AVX instructions through C functions (called intrinsic functions). Intrinsics are replaced directly to vector instructions without the overhead of function calls. In this paper, we achieved better performance when compared to compiler generated vectorization of our computational kernels.

MPI Graph Topology: Given a process interaction graph, MPI provides support to map the processes on a compute cluster. The application level topology can be mapped to the the physical topology of a network using cartesian and graph topology functions in MPI. Since a good mapping of processes to network topology reduces the data communication volume across the network, we have used graph topology functions in our implementation.

D. Related Work

With the volume of data increasing due to its spatio-temporal nature, parallelization of existing algorithms have been done [9], [10], [13], [19]. Existing approaches use spatial partitioning methods like quadtree for parallelization [10].

GPU-based implementations of geospatial filter-based algorithms have been presented in [11], [12]. MPI-based parallelization of geospatial polygon overlay and spatial join has been presented in [20], [21], [26].

A Matlab-based shared memory parallelization has been described in [9]. Hadoop MapReduce has been used to parallelize Getis-Ord based Hotspots detection problem using quadtree-based decomposition of spatial data [10]. Apache Spark framework has also been used to parallelize spatial hotspot computation [13], [19]. Spark MapReduce papers are short papers from GIS Cup competition organized with SIGSPATIAL conference [13], [19]. Hadoop and Spark based projects make good use of thread-level and coarse-grained parallelism but do not take full advantage of HPC resources (e.g., SIMD, GPUs) thus leaving performance on the table [10], [13], [19]. The trade-offs of calculating weight matrix vs on the fly computation has been discussed in .

Compared to related literature, our paper further explores additional hardware and software parallelization opportunities. GPU SIMT parallelization and CPU SIMD parallelization along with communication optimizations are the novelties compared to related literature.

III. Parallel formulation of spatial autocorrelation

We can use Getis-Ord algorithm to calculate the G_i^* statistic for each feature in a dataset [16]. In geospatial analysis, it gives a Z-score statistic G_i^* where x_j is the value for polygon j. $w_{i,j}$ is a weight parameter between polygons i and j which is inversely proportional to the active distance between them. N is equal to the total number of polygons in our dataset. Positive and negative G_i^* values denote hot and cold spots

respectively and the absolute value of G_i^* is proportional to the intensity of clustering for the i^{th} polygon.

The equations to the Getis-Ord algorithm are as follows:

$$\overline{X} = \frac{\sum_{j=1}^{n} x_j}{n} \tag{1}$$

$$\overline{X^2} = \frac{\sum_{j=1}^{n} x_j^2}{n} \tag{2}$$

$$S_X = \sqrt{(\overline{X^2}) - (\overline{X})^2} \tag{3}$$

$$W_{X_i} = \sum_{j=1}^{n} w_{i,j} x_j \tag{4}$$

$$W_i = \sum_{j=1}^{n} w_{i,j} \tag{5}$$

$$W_i^2 = \sum_{j=1}^{n} w_{i,j}^2 \tag{6}$$

$$S_i = \sqrt{\frac{\left[n * W_i^2 - \left(W_i\right)^2\right]}{n-1}} \tag{7}$$

$$G_i^* = \frac{W_{X_i} - \overline{X} * W_i}{S_X * S_i} \tag{8}$$

For Moran's I:

$$W = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j} \tag{9}$$

$$I = \frac{n}{W} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j}(x_i - \overline{X})(x_j - \overline{X})}{\sum_{i=1}^{n}(x_i - \overline{X})^2} \tag{10}$$

Values of I usually range from -1 to $+1$. Values significantly below $(1-N)^{-1}$ indicate negative spatial autocorrelation and values significantly above $(1-N)^{-1}$ indicate positive spatial autocorrelation. For statistical hypothesis testing, Moran's I values can be then transformed to z-scores.

Geary's C:

$$C = \frac{n-1}{2W} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j}(x_i - x_j)^2}{\sum_{i=1}^{n}(x_i - \overline{X})^2} \tag{11}$$

N is the number of spatial units indexed by i and j. x is the variable of interest; $\bar{x}$ is the mean of x; $w_{i,j}$ is a matrix of spatial weights with zeroes on the diagonal (i.e., $w_{ii} = 0$ and W is the sum of all $w_{i,j}$.

The value of Geary's C lies between 0 and some unspecified value greater than 1, usually lower than 2. Values significantly lower than 1 demonstrate increasing positive spatial autocorrelation. Values significantly higher than 1 illustrate increasing negative spatial autocorrelation. Geary's C is inversely related to Moran's I. Moran's I is a measure of global spatial autocorrelation, while Geary's C is more sensitive to local spatial autocorrelation.

A. Algorithm

The Algorithm for Getis-Ord is as follows:

1) Load all the Points and their x attribute values.
2) Calculate the mean of all the x values, denoted by $\overline{X}$.
3) Calculate the mean of all the x^2 values, denoted by $\overline{X^2}$.
4) Calculate S, the standard deviation of all the x values.
5) Calculate the values for $w_{i,j}$, the weight metric between polygon i and polygon j.
6) Calculate $w_{i,j}^2$ from $w_{i,j}$.
7) For each i, calculate W_i from $w_{i,j}$.
8) For each i, calculate W_i^2 from $w_{i,j}^2$.
9) For each i, calculate S_i from W_i and W_i^2.
10) For each i, calculate W_{X_i} from $w_{i,j}$ and x values.
11) For each i, calculate G_i^*.

B. Complexity

The time complexity of this algorithm is $O(N^2)$ and the space complexity of this algorithm is $O(N)$. This analysis of time complexity is contingent on the assumption that inverse distance squared (impedance) is used for $w_{i,j}$ and any similar $O(c)$ method of calculating $w_{i,j}$ would keep the analysis the same. Similarly, for the space complexity no pre-calculations of $w_{i,j}$ are assumed. Pre-calculations of $w_{i,j}$s would make the space complexity to become $O(N^2)$ too.

C. Weight Matrix

The most common technique of calculating $w_{i,j}$ is the metric called the inverse distance. Distance could be different types but most typically the euclidean distance. Inverse distance is a metric would be a high value for things that are closer and low value for things that are spatially further apart. It should be noted that $w_{i,j} = k \ \forall (i = j)$, where k is a value of no consequence and is just used as a placeholder because in this case both i, j would be the same point so no distance and undefined inverse distance. On, the other end, objects further than a certain threshold can be deemed to have a inverse distance value of zero i.e. $w_{i,j} = 0$ if $invDist(i,j) < \epsilon$. Also, $w_{i,j} = w_{j,i}$ because both are distance-based quantities which does not vary on direction. Hence, if w was to be modeled as a matrix, it would be a $n \times n$ symmetric matrix with diagonals all k. Basically, it is an adjacency matrix where $w_{i,j}$ corresponds to the weight, as it relates to the spatial relation between two areas i and j.

D. Spatial Sorting

Spatial sorting is used to arrange 2-dimensional points in 1-dimensional order based on spatial proximity (locality). Space filling curves are used for spatial sorting, such as Z-order [15] and H-order (also known as Hilbert curve). For illustration, let us assume that we have a list of tuples, where the first entry is the x-coordinate and the second entry is the y-coordinate of a point. After sorting the list spatially, points that are closer to each other in the xy plane would appear closer in the list. Proximity of the points - difference in their index values in the sorted list would be an indication of proximity of the points in euclidean space and vice versa.

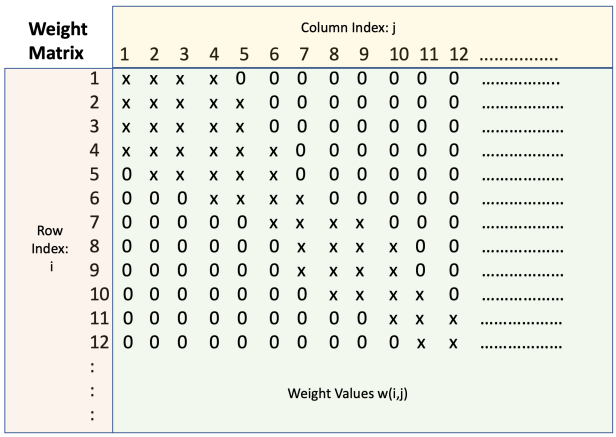

Fig. 3. Slice of the Weight Matrix. Each row and column index corresponds to a polygon id. For any two polygons i and j, element at index (i, j) is the inverse of the euclidean distance between centroids of i and j.

Having the polygons from our data sorted has special implications for our application and acceleration objectives, especially the affect it has on the weight matrix. Looking at Figure 3, we can observe that if the polygons are spatially sorted, then in each row i, the columns that have non-zero entries are only the columns numbered close to the value of i. This is because, as polygons get further apart, their inverse distance decreases and beyond a threshold, they simply become zero. So, for each row i, the columns j for whose values are further apart, their values are simply zero because it represents the underlying property that polygon i and j are just spatially further away from each other.

Expanding upon this property, we will find that for each row i there are only columns in the range $(i - l_i, i + r_i)$ for which the weight values are non-zero. Let l_i be the number of entries to the left of i that are non-zero and r_i be the number of entries to the right of i that are non-zero. Given a large map with lots of polygons, the range $(l_i + r_i)$ can become significantly small, making our matrix a sparse matrix with only elements around the main diagonal being non-zero and elements further away from the diagonal being mostly zeros. For example, with 100k polygons the max range $(l_i + r_i)$ was less than 200.

Furthermore, for the rapid recalculation part, in events where we only have new data for a few polygons and we want to update the scores, the only polygons that require recalculation would be the polygons which have new data and the polygons with which it has a non-zero weight relationship.

Comparison with R-tree: An alternative to using the weight matrix would be the use of a R-tree like approach. Here, our cutoff threshold ϵ from the weight matrix would translate to a certain distance and we would then query the tree to get all polygons within that distance range from the query polygon. We could then calculate weights $w_{i,j}$ for each query polygon i and queried polygons denoted by j. If we use this approach, rather than the sorting and pre-calculating weights, then it would add overheads needed to build a tree. This is in contrast to the tradeoff of sorting all the polygons. Since the locations of the polygons are static, the tree would only be needed to be built once just like the sorting. The advantage of using weight matrix is that the weights will be available in memory easily accessible for SIMD operations. Also, in the cases of the square tiles, sorting is extremely efficient and building a tree would just be an overhead. In an R-tree approach, each polygon will be able to query its list of neighbours and then calculate the corresponding weights with each neighbour. Since the polygons will be unsorted, each weight calculation will access arbitrary areas of the memory and no cache-based gain will be achieved. Also, using a vectorized approach will not be possible without further sorting and ordering because the results of the query may not be in a contiguous memory. The distinct advantage of using R-trees can be that their build cost is not high, their query can be easily parallelizable and storing the weight matrix might not be necessary.

IV. Acceleration Techniques

A. Cache Access Optimization

We have three arrays of size N – two are arrays that have the x-location and y-location for each point, and another is an array of attribute values of each point. Let's denote the first two arrays by p and the next array by x. We need to fill a 2D array of size $n \times n$ with $w_{i,j}$s. Let's call this array w. Assuming there is a cache block size of B, whenever calculating any $w_{i,j}$, we get two B blocks of p and one B block of w loaded into the cache, so in this case, instead of linearly calculating the values of w, we calculate all the combination of $w_{i,j}$ that we can from these two blocks of p in an order where we can write into the loaded B block of w. Once we have a filled $w_{i,j}$ matrix array, whenever looping through it, we need to make sure that we access it in the proper order. Looping through $\sum_{j=1}^{n} w_{i,j}$ for a fixed i might be expensive in column-major architectures than looping through $\sum_{j=1}^{n} w_{j,i}$ but since $w_{i,j} = w_{j,i}$ doing both will give the same result.

B. Weight Matrix Storage Optimization

Since the weight matrix is symmetric, we can store only the upper triangular matrix. Furthermore, since the non-zero values are only near the diagonal we would only need to store at most $maxr = \forall_i \max r_i$ values for each polygon. So, in the worst case, the weight matrix would need $n * maxr$ space compared to its n^2 size. But this approach makes SIMD operations inefficient because we would need to index up or down to find the neighbours to the left of polygon i. Due to symmetry,n^2 and $maxl = \forall_i \max l_i$ would be equal. So, we could store a $n * (2 * maxr)$ array, which is still better than the n^2 array. Here the N rows will be the polygons and $(2 * maxr)$ columns would be weight with the non-zero neighbours. This way, although the storage is doubled from the most compressed form, being able to access a contiguous memory of weights will significantly improve the cache access and make SIMD operations easily accessible. Furthermore, if the weight matrix is now stored in a file, then, that too can be

easily read with contiguous memory access and the amount needed to be read by each process decreases significantly, almost by a factor of $n/maxr$.

C. OpenMP Parallelization

OpenMP parallelization is based on the equations of the Getis-Ord algorithm as shown earlier. The steps from Getis-Ord algorithm III-A, Step 2, 3 and 5 were parallelized using parallel loops with reduction. All the steps, including calculating each of the G_i^*, are parallelized. If recalculation of results is not required, then steps 5 through 10 can be parallelized to run by each thread for each polygon i along with a second level of parallelism inside the loop for calculating all the sums and G_i^* values.

D. OpenACC Parallelization

OpenACC compiler pragmas support both CPU and GPU parallelization. We have used OpenACC for GPU parallelization. Compared to OpenMP, additional steps include data copy to GPU (in and out). We have used reduction pragma in OpenACC for additions. For example, in Algorithm III-A, Step 1, once the x values are copied to the GPU, for Steps 2 and 3, we can do reductions to get the summation results. Only the output G_i^* values are copied back to the host CPU. Our OpenACC implementation leverages our existing C/C++ code.

E. CUDA Parallelization

We have also used CUDA for GPU parallelization of our kernels. Compared to OpenACC, CUDA gives more control in using the GPU. For algorithm III-A, we added CUDA kernels for each steps. For large datasets that do not fit in the GPU memory, especially the weight matrix whose size grows quadratically in the number of inputs, we do calculations in batches by moving data in and out of the GPU. Data movement between GPU and Host can be an expensive step compared to computation especially when done multiple times.

F. MPI Graph Topology (Distributed Memory)

Using MPI, process ids are used to split the data among multiple compute nodes for a distributed memory parallelization. We use allreduce collective function to merge the partial results from Steps 2 and 3 of algorithm III-A. We need to broadcast the reduced values to all the ranks as well. Also, for Step 5, each polygon needs to calculate the $w_{i,j}$ values and the MPI ranks need communication to share the location information. We assign a MPI rank to each polygon. This process mapping scheme helps in creating better MPI process topology, which we discuss next.

Given the nature of weights which decays with increasing distance, polygons that are further from each other have a weight of zero. This means that only polygons that are close to each other need to communicate with each other. The Weight matrix can then be utilized to create an adjacency matrix (for graph) where entries in this new matrix are 1, if the weights are greater than zero, and zero otherwise. We translate this polygon adjacency matrix to MPI processes adjacency matrix for each process as required by Graph Topology function in MPI. MPI has methods that can take this adjacency matrix and arrange processes in such a way that minimizes the amount of communication among processes. We have used the following function for graph topology in MPI.

```
MPI_Dist_graph_create_adjacent(
    MPI_COMM_WORLD, degree, neighbours,
    MPI_UNWEIGHTED, degree, neighbours,
    MPI_UNWEIGHTED, MPI_INFO_NULL, 1,
    &new_dist_comm);
```

Listing 1. Adjacent distributed graph creation

Since the weight matrix is symmetric, the indegrees are equal to the outdegrees and the sources are same as the destinations. We have used MPI_UNWEIGHTED because the volume of communication is the same when communication takes place. It is important to set reorder equal to 1, if we want MPI to figure out the best configuration to reduce the amount of cross-node communication. Setting reorder to be true, means that in the new MPI_Comm, the ranks of MPI processes will be different from the global ranks in MPI_COMM_WORLD. Hence, to avoid double loading of the input data (before and after process reordering), we divide the overall data loadin into two stages. In the first stage we load partial data that is necessary and then load all the other remaining data only after this reorder has taken place. This is efficient and it also ensures that MPI processes will not have data corresponding to their old ranks.

G. Communication Efficiency on Distributed Memory

If we have P processes, each process will have N/P polygons and each of them will have to calculate N/P G_i^* values. However, $\overline{X}$ and S are the same for N polygons. So, each N/P process have to calculate those values only once. $\overline{X}$ and S are simply mean and standard deviation, and we can use any of the existing communication efficient algorithms to calculate those. The main communication bottleneck here is that for each polygon i to calculate G_i^*, it needs $w_{i,j}$ and x_j for all N js which means P all-to-all communication steps which is $O(P^2)$ communications. Each broadcast would have to send the appropriate x_j values along with parameters to calculate $w_{i,j}$ values. Using graph topology built on top of a weight matrix that preserves neighborhood information for each MPI process, the communication can be potentially optimized to $O(P)$ communication steps.

H. Vectorization with compiler intrinsics

For single precision floating point data type (32 bits), 8-way parallelism can be potentially exploited by using 256 bit vector register supported by Advanced Vector Extensions (AVX) [8]. AVX-512 intrinsics can support 16-way parallelism because of wider SIMD registers. Intrinsic functions work like inline functions. There is no overhead of function calls because compilers replace these functions with corresponding vector assembly instructions. Our implementation of equations 8, 1 and 3 is geared towards exploiting vectorization via intrinsics.

Arithmetic (summations, multiplications, etc), data movement (load/store), and comparison operations are fully vectorized. The denominator and numerator terms for equation 8 are also vectorized efficiently.

In Algorithm 1, we show an example of using advanced vector intrinsics to calculate the weight matrix using the inverse euclidean distance and setting all weight values below threshold epsilon (epi) to be zero. Broadcast function is used to set all the elements of a SIMD register with the same value that was passed to it as an argument. Please refer to [8] for details on the functions used here.

It can be seen that the code is optimized enough to start vector operations always at aligned memory for each i loop using the second j loop and control variable k. Also, the code only does one calculation for $w_{i,j}$ and $w_{j,i}$ values because they are the same due to symmetry. There is a post-processing step done after this to fill the $w_{j,i}$ values. This will ensure that whenever we need $w[i]$ for any polygon i, we will have the full contiguous memory of size N with values for all $w_{i,j}$.

Algorithm 1 Intrinsics based algorithm for calculating weights

Input: N, cutoff value epi
Output: populated weights w

```
1: declare __m256 epis, x1, x2, xx, y1, y2, yy, z
2: declare int i, j, k and assign k ← 8
3: epis ← mm256_broadcast_ss(epi)
4: for (i = 0; i < N; i++) do
5:    for (j = i + 1; j < k; j++) do
6:       w[i*N + j] ← invEucDist(x, y, i, j, epi)
7:    end for
8:    for (j = k; j < N; j = j + 8) do
9:       x1 ← mm256_broadcast_ss( x + i )
10:      x2 ← mm256_load_ps( x + j )
11:      xx ← mm256_sub_ps( x2, x1 )
12:      xx ← mm256_mul_ps( xx, xx )
13:      y1 ← mm256_broadcast_ss( y + i )
14:      y2 ← mm256_load_ps( y + j )
15:      yy ← mm256_sub_ps( y2, y1 )
16:      yy ← mm256_mul_ps( yy, yy )
17:      z ← mm256_add_ps( xx, yy )
18:      z ← mm256_rsqrt_ps( z )
19:      // SIMD compare if z > epis
20:      bmask ← mm256_cmp_ps(z, epis, CMP_GT_OQ )
21:      z ← mm256_and_ps(z, bmask) // (z & bmask)
22:      mm256_store_ps( w + i*N + j, z )
23:   end for
24:   k ← (((i + 1)/8) + 1) * 8
25: end for
```

I. OpenMP & Vectorization

On top of our vectorized code, we added thread-level data parallelism using OpenMP to leverage multiple vector units available on modern multi-core CPUs. For this combined parallelization, cache and register memory availability with multiple parallel threads are the main issues. With reference to code, algorithm 1, the approach that gave us the most benefit was to run the i loop in OpenMP parallel regions while maintaining contiguous data access for each thread. If t is the number of OpenMP parallel threads, this can be achieved with using a guided OpenMP schedule with chunk size ck such that $1 < ck < (N/t)$. Having a lower value of ck will split the iterations into threads in such a way that the first among the earlier threads will have the largest chunk size and less memory access overhead, but later threads will have smaller chunks size and higher cache overhead. Also, with multi-threading, it is necessary to keep in mind that depending on the processor, each core will have only a limited number of SIMD registers (usually 32) and limited L1 cache size, so choosing a thread count t that does not overwork each core is necessary to see any benefits from the combined acceleration approach.

J. MPI & Multiple GPU (CUDA)

If multiple nodes with GPU are available, then MPI can be used to offload much of the processing to the GPUs by combining the MPI and CUDA codes. Once each MPI process has the data it is going to be processing, it can easily copy it to GPU device and get results. This will work even if there are multiple MPI processes running in the node. Even if each node has multiple GPUs, MPI processes can use their rank to select one of the available GPUs and offload their computation. This has been shown in Figure 4. If there are multiple nodes each with multiple GPUs, this same approach will work with the combined MPI. The best way to use MPI with CUDA is to have a separate cuda file with extern C functions that are capable of executing the cuda kernels. This has been demonstrated in Figure 5. Pointer to the data structures from the host's main memory can be passed into this function with useful information like the rank of the MPI process that's calling it. Using cudaGetDeviceCount, cudaSetDevice and the MPI rank, the function can call the kernel and copy back the memory after computation to host using the host pointers. Figures 4 and 5 are only for demonstration purpose and show a case where a node has multiple GPUs and number of MPI processes are equal to the number of GPUs per node.

If multiple GPUs are going to be used in a node, it is also a good idea to minimize all cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost to because that is the step that consumes the most time. So, a preprocessing step to allocate memory on the GPUs and passing back the device pointers to host to use in further calculations is recommended.

K. Rapid Recalculation

Even in scenarios where the data emerges or changes at certain time intervals, the location based data and spatial relationships remains constant. For example, in the COVID data cases, the number of daily cases would be different but the distance between two counties would remain the same. So, whenever we would need to re-calculate the results, we would need to only recalculate some of the equation, i.e. the equations that are dependent on x. The equations independent

```
int rank, nprocs, ngpus;
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
num_gpus(&ngpus);

int length = get_length();
int clen = length/ngpus;
int start = rank*clen;
float *Gi = (float*)malloc(clen*sizeof(float));
float *Wxi = loadWxi(start,clen);
float *Wi = loadWi(start,clen);
float *Si = loadSi(start,clen);
float xbar = loadxbar();
float Sx = loadSx();

launch_calcGstar(Gi, Wxi, Wi, Si, &xbar, &Sx, clen, rank);
MPI_Barrier(MPI_COMM_WORLD);

float *all_Gi = NULL;
if (rank == 0)
{
    float *all_Gi = (float*)malloc(length*sizeof(float));
}
MPI_Gather(Gi, clen, MPI_Float, all_Gi, clen, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Fig. 4. MPI part of multi-gpu

```
__global__ void __calculate__ (float *Gi, float *Wxi, float *Wi,
                               float *Si, float *xbar, float *Sx)
{
    const int i = threadIdx.x + blockIdx.x * blockDim.x;
    Gi[i] = (Wxi[i] - (xbar[0] * Wi[i])) / (Sx[0] * Si[i]);
}

extern "C" void num_gpus(int *num)
{
    cudaGetDeviceCount(num);
}

extern "C" void launch_calcGstar(float *Gi, float *Wxi, float *Wi,
                                 float *Si, float *xbar, float *Sx,
                                 int length, int rank)
{

    cudaSetDevice(rank);

    int ARRAY_SIZE = length;
    int FLOAT_SIZE = sizeof(float);
    int ARRAY_BYTES = ARRAY_SIZE * FLOAT_SIZE;

    float *Gi_gpu,*Wxi_gpu,*Wi_gpu,*Si_gpu,*xbar_gpu,*Sx_gpu;

    cudaMalloc((void **)&Gi_gpu, ARRAY_BYTES);
    cudaMalloc((void **)&Wxi_gpu, ARRAY_BYTES);
    cudaMalloc((void **)&Wi_gpu, ARRAY_BYTES);
    cudaMalloc((void **)&Si_gpu, ARRAY_BYTES);
    cudaMalloc((void **)&xbar_gpu, FLOAT_SIZE);
    cudaMalloc((void **)&Sx_gpu, FLOAT_SIZE);

    cudaMemcpy(Wxi_gpu, Wxi, ARRAY_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(Wi_gpu, Wi, ARRAY_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(Si_gpu, Si, ARRAY_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(xbar_gpu, xbar, FLOAT_SIZE, cudaMemcpyHostToDevice);
    cudaMemcpy(Sx_gpu, Sx, FLOAT_SIZE, cudaMemcpyHostToDevice);

    __calculate__ <<<1,ARRAY_SIZE>>> (Gi_gpu,Wxi_gpu,Wi_gpu,Si_gpu,xbar_gpu,Sx_gpu);
    cudaDeviceSynchronize();

    cudaMemcpy(Gi, Gi_gpu, ARRAY_BYTES, cudaMemcpyDeviceToHost);

    cudaFree(Gi_gpu);
    cudaFree(Wxi_gpu);
    cudaFree(Wi_gpu);
    cudaFree(Si_gpu);
    cudaFree(xbar_gpu);
    cudaFree(Sx_gpu);

}
```

Fig. 5. CUDA part of multi-gpu

of x could be pre-calculated and stored for easy access and retrieval. The equations independent of x in equation 8 for G_i^* are equation 5 for W_i and equation 7 for S_i and their dependent equations. Hence, for each polygon, W_i and S_i remain unchanged for newer values of x and do not need to be recalculated from the beginning.

Next, lets consider a boundary case where we have a new value for only one polygon and there is change in only one value of x. In such a case, the global values of $\overline{X}$ and S_x would change and would need to be updated across all polygons. However, we would only need to recalculate W_{X_i} for cases $w_{i,j} \neq 0, j = k$ where x_k is the existing polygon value and Δx_k is the change in value for x_k.

So the equations become

$$\overline{X}_{new} = \overline{X} + \frac{\Delta x_k}{n} \tag{12}$$

$$\overline{X^2}_{new} = \overline{X^2} + \frac{2 * x_k * \Delta x_k + \Delta x_k^2}{n} \tag{13}$$

$$\begin{aligned} {S_X}^2_{new} &= \overline{X^2}_{new} - (\overline{X}_{new})^2 \\ &= \overline{X^2} + \tfrac{2*x_k*\Delta x_k+\Delta x_k^2}{n} - (\overline{X} + \tfrac{\Delta x_k}{n})^2 \\ &= \overline{X^2} + \tfrac{2*x_k*\Delta x_k+\Delta x_k^2}{n} - (\overline{X})^2 - (\tfrac{2*\overline{X}*\Delta x_k}{n}) - (\tfrac{\Delta x_k}{n})^2 \\ &= S_X^2 + \tfrac{2*x_k*\Delta x_k - 2*\overline{X}*\Delta x_k}{n} \end{aligned}$$

$$ {S_X}^2_{new} = S_X^2 + \frac{(2 * \Delta x_k) * (x_k - \overline{X})}{n} \tag{14}$$

$$W_{X_i new} = W_{X_i} + w_{i,k}\Delta x_k \tag{15}$$

Next, lets consider the general case where there are multiple new x values for multiple polygons. In this case, we would only need to recalculate W_{X_i} for cases $w_{i,j} \neq 0 \ \forall \ j = k$ where x_ks are the updated polygon values. In this case, the equations become:

$$\overline{X}_{new} = \overline{X} + \frac{1}{n}\sum_k \Delta x_k \tag{16}$$

$$\overline{X^2}_{new} = \overline{X^2} + \frac{1}{n}\sum_k (2 * x_k * \Delta x_k + \Delta x_k^2) \tag{17}$$

$$ {S_X}^2_{new} = S_X^2 + \frac{2}{n} * \sum_k (\Delta x_k * (x_k - \overline{X})) \tag{18}$$

$$W_{X_i new} = W_{X_i} + \sum_k w_{i,k}\Delta x_k \tag{19}$$

Hence, if there are only few polygons with updated values, and if we have pre-calculated values from previous iterations, then we can calculate the difference and use the difference to reduce a lot of recalculations. For example, if there were only 100s of counties that had updated data from the previous day, then we could rerun calculations for just those 100 and update the G_i^* values. Also, note that Δx_k values can be negative too, in case of decrease in x values.

V. Experimental Results

For the experiments, both real world data and simulated/-generated data were used to test the implementations.

A. Real World COVID Data

One of the primary motivation for this work was to track COVID hotspots, especially as they were emerging and altering. One of the main sources of COVID related data was the United States Center for Disease Control. Different geographic level (like cities, districts, county, states) based data on daily reported values are available. This data had necessary COVID related statistics like active cases, new cases, closed cases, deaths, recovered etc. However, for geospatial analysis, we require geographic data too. For the experimental timing results provided in this paper, we focused on the county level analysis. Geographic data required are county locations and boundaries. This information was available from the Census Bureau's MAF/TIGER geographic database U.S. County Boundaries TIGER dataset [5]. For autocorrelation calculations, we require only certain properties from the geographic data. For each county, we required its boundary information to calculate its centroid. This centroid information was further used to calculate the inverse distance for the weight values among county polygons. Next, we needed to match the county polygons with its corresponding COVID data. Counties have unique identifiers called GEOID, so each of these county polygons had a unique five digit identifier known as the FIPS code. Also, the county level COVID data along with each county information had a corresponding FIPS code. This common unique id made it easier to join the COVID data with the geographic data. The counties geographic data had 3,233 polygons along with other data entities of which the extra unnecessary information were discarded and this was processed to get a dataset with the following entities: County, State, FIPS, and Centroid. Then for each date, the entities for the available COVID data were: Date, County, State, FIPS, new cases, active cases, recovered cases, total cases, new deaths, and total deaths.

B. Simulated/Generated Datasets

Simulated data were generated mostly for the unit point incidence data and the aggregated point incidence data. The data was generated randomly. For the unit point incidence data, the sample space was divided into a uniform square grid, and each square cell was considered as the polygon for that region. Next, the centroid for each of the square tile was calculated. Then, using different random distributions, x-values (attributes) were assigned to each square tile. The x-values were used to simulate the count of events inside the square tile. Finally the data entities for each square tile were: id, centroid, x1, x2, x3, ..., xn. Using the centroid values to calculate the inverse distances among the square tiles, the weight matrix was populated.

For the aggregated point incidence data, first location for the aggregation points were generated from a uniform random distribution across the sample space. Then a fast Voronoi boundary calculation [18] was used to generate the boundaries for each unit point. These boundaries represented the polygon for that region and the aggregation points were used as centroids for that region. Next, similar to data generation with the square grids, different random distributions were used to simulate x-values which were assigned to each polygon. Finally the data entities for each aggregation points polygon was: id, centroid, x1, x2, x3, ..., xn. Using the centroid values to calculate the inverse distances among the polygon boundaries, the weight matrix was populated.

C. Hardware Description

Experiments were performed on two machines with the following hardware configurations. Machine 1 (M1) has two Intel Xeon E5 v4 CPUs (2.10 GHz), where each CPU has 18 cores (36 thread). M1 has 500 GBs of RAM. M1 also has an Nvidia TITAN V GPU with 5120 CUDA cores. On the Intel Xeon E5, there is L1 cache of 32KB per core. L2 and L3 cache sizes are 256 KB and 2.5 MB. L2 cache is per core. L3 cache is per NUMA node. The gcc verision is 4.8.5, nvcc is V11.2.67 and pgcc is 21.2.0.

Machine 2 (M2) is a medium sized compute cluster with multiple nodes used for running experiments with a scheduler. Compute nodes in M2 contains AMD Rome which is a 64 core (128 thread) CPU with a base frequency of 2 GHz, NVIDIA Tesla V100 GPUs which has 5120 CUDA cores at base frequency of 1.20 GHz and 512 GBs RAM. Compute nodes and storage are connected via a 100 GB/s Infiniband network. On the AMD Rome, there is L1 instruction cache of 32KB per core and similarly L1 data cache of 32KB per core. There is mid-level cache (MLC) or L2 of 512 KB per core. AMD Rome has 16 x 16 MB L3 cache which is the last level cache and is a shared cache of 16 MB per 4 core. The gcc version is 9.2.0, mpi is mvapich2, nvcc is V11.2.152 and pgcc is 21.11.0.

D. Performance Engineering Results

Table I show the aggregation of speedup gained from different methods from multiple experiments at different data sizes. Every acceleration method improves the computation speedup and combining different approaches has even greater yield. For OpenMP and MPI, the shown speedup holds as long as the threadCount or numProcess is less than the number of cores.

The AVX2 codes were implemented in both Intel and AMD CPUs and the gain in performance was similar across both. Because 8 single precision floating point variables can be loaded in 256 bits of a SIMD register, there is potentially 8-way SIMD parallelism that can be exploited when compared to scalar code. We observe upto 6x speedup using SIMD-optimized code. We used linux perf tool to measure the impact of improved vectorization through intrinsic functions on x86 processors. An analysis through the perf tool showed that with intrinsics the number of CPU cycles were reduced by a factor of almost 40x while the instructions per cycle (IPC) doubled. Higher IPC value represents better CPU utilization.

TABLE I
PARALLELIZATION METHOD AND CORRESPONDING BEST SPEEDUP (25K DATASET)

Parallelization	Speedup
GPU CUDA (single node)	100×
GPU OpenACC (single node)	100×
OpenMP (16 thread)	15.4×
AVX2 intrinsics	6×
AVX2 + OpenMP	90×
MPI (16p)	15×
MPI (16p) + AVX2	90×
MPI (8 gpu nodes) + CUDA	750×
MPI (4 gpu nodes) + CUDA	380×

Also, the number of branches decreased by almost 50x while branch misses reduced by 1.5x. This is attributed to the advantages of loop unrolling on line number 8 of Algorithm 1 (loop variable j is incremented by 8). Reduction in branch misses leads to higher instruction level parallelism through instruction pipelining because of reduction in control hazards. Furthermore, cache loads decreased by 16x and cache misses decreased by more than 2x.

From a vectorization perspective, the difference in performance is attributed to the choice of SIMD registers and vector instructions selected by the compiler with/without intrinsics. We used GCC compiler with -O3 flag to enable compiler auto-vectorization. In compiler generated code, XMM registers with 128 bits width were used for critical parts of the kernel. In the version with intrinsics, compiler generated code had YMM registers with 256 bits width. Wider registers have the benefit of packing more data elements in a single register. We looked at the assembly code generated with/without intrinsics using double precision floating point data. For data movement, *vmovsd* was generated in the sub-optimal code instead of *vmovapd*. s stands for scalar in vmovsd. p stands for packed in *vmovapd*. Similarly, *vmulsd* was generated by compiler in the suboptimal code instead of *vmulpd*.

Figure 6 shows the time (in log_2 scale) for different sizes of data. Average from multiple runs of the experiments are shown. The best implementation remains the MPI+CUDA approach.

Execution times from an experiment with 300,000 polygons are shown in Table III. Using a non-optimized sequential C code, it takes about 36 minutes to run from start to finish. The computationally intensive parts can be divided into three parts. First part is the spatial sorting. Second part is calculating and populating the weight matrix. Final part is calculating all G_i^* values. The above mentioned speedups in Table I are mostly gained in the second and third parts. OpenACC and CUDA brings down 780 seconds to calculate the weight matrix down to about 9 seconds. AVX2 intrinsics brings it down to almost 110 seconds. Adding OpenMP parallelization to AVX2, with a thread count of 16 threads brings the time down to almost 7 seconds and its performance is very similar to that of MPI. The MPI+CUDA results is using 4 GPUs concurrently which is the fastest. MPI+CUDA took 2 seconds. Table II shows the average speedup and efficiency of using multiple OpenMP threads.

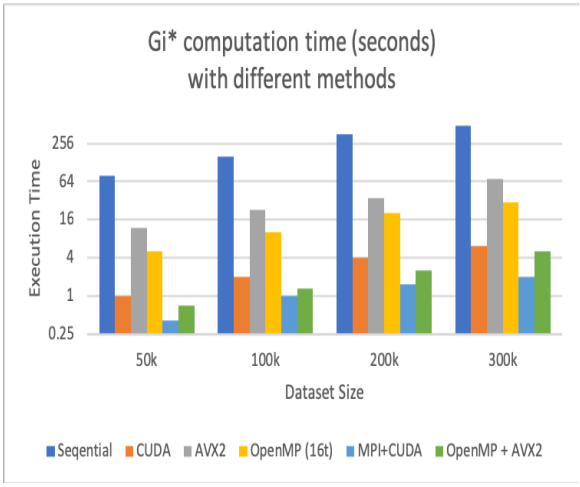

Fig. 6. Comparison at different data sizes. OpenMP version is running on 16 threads.

TABLE II
OPENMP SPEEDUP AND EFFICIENCY

Threads	Avg Speedup	Speedup/thread
2	1.9	0.950
4	3.7	0.925
8	7.7	0.963
16	15.4	0.963
32	30.1	0.941

TABLE III
AVERAGE EXECUTION TIMES FOR 300K POLYGONS

Method	Sorting	Wmatrix	G_i^*	Total (minutes)
Sequential	900s	780s	480s	36
CUDA	10s	9s	6s	0.42
AVX2	500s	110s	69s	11.4
OpenMP (16 t)	150s	51s	30s	4
MPI+CUDA	10s	2s	2s	0.24
OpenMP+AVX2	150s	7s	5s	2.7

TABLE IV
RTREE BASED TIMES FOR 300K POLYGONS

	Building	Querying	G_i^*	Total (minutes)
Sequential (No Sort)	20s	60s	520s	10
OpenMP (16 t)	20s	4s	37s	1
OpenMP+AVX2	20s	4s	10s	0.6

Table IV shows R-tree based execution time for 300K polygons. This sequential version performs better than the version with spatial sorting because of R-tree data structure. This version does not use spatial sorting, as shown in Table IV. OpenMP parallelization speeds up query operations and calculation of G_i^* values compared to the sequential baseline. SIMD parallelization using AVX2 is applied to G_i^* calculations only.

The best performance on a single compute node is by using 16 threads accelerated by AVX2 SIMD extensions.

Table V shows the use of acceleration and rapid recalculation techniques applied to calculate daily G_i^* values for the US Counties using real world COVID data for 500 days to see the evolution of the spread of infection over the time period.

TABLE V
500 DAYS TIME SERIES G_i^* CALCULATION FOR REAL US COUNTIES DAILY COVID DATA [5] [4]

Method	Time (minutes)
Sequential	33
CUDA	0.5
AVX2	6
OpenMP (16t)	3
MPI+CUDA	0.3
OpenMP+AVX2	1

VI. Conclusion and Future Direction

We have demonstrated successful acceleration of spatial autocorrelation kernel. This acceleration can be used for industrial and scientific application requiring faster solutions and the techniques mentioned in the paper can be transferred to apply to wide variety of similar statistical kernels. Future directions of this work can be extending the rapid recalculation work for streaming and online real-time solutions and expanding the scope of the work for cloud infrastructures where different acceleration techniques are combined to automatically achieve the best acceleration depending on hardware configuration and availability.

Acknowledgment

This research used the Raj high-performance computing facility funded by the National Science foundation award CNS-1828649 and Marquette University.

References

[1] https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-statistics/h-how-hot-spot-analysis-getis-ord-gi-spatial-stati.htm.

[2] Marc P Armstrong and Richard Marciano. Massively parallel processing of spatial statistics. *International Journal of Geographical Information Systems*, 9(2):169–189, 1995.

[3] Marc P Armstrong, Claire E Pavlik, and Richard Marciano. Parallel processing of spatial statistics. *Computers & Geosciences*, 20(2):91–104, 1994.

[4] https://covid.cdc.gov/covid-data-tracker/index.html.

[5] https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html.

[6] http://resources.esri.com/help/9.3/ArcGISEngine/java/Gp_ToolRef/Spatial_Statistics_tools/how_hot_spot_analysis_colon_getis_ord_gi_star_spatial_statistics_works.htm.

[7] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[8] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[9] Mingjun Li. MS Thesis: A Parallel Algorithm and Implementation to Compute Spatial Autocorrelation (Hotspot) Using MATLAB. *MS Thesis*, 2020.

[10] Yan Liu, Kaichao Wu, Shaowen Wang, Yanli Zhao, and Qian Huang. A mapreduce approach to gi(d) spatial statistic. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, pages 11–18, 2010.

[11] Yiming Liu and Satish Puri. Efficient filters for geometric intersection computations using gpu. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, pages 487–496, 2020.

[12] Yiming Liu, Jie Yang, and Satish Puri. Hierarchical filter and refinement system over large polygonal datasets on cpu-gpu. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 141–151. IEEE, 2019.

[13] Paras Mehta, Christian Windolf, and Agnès Voisard. Spatio-temporal hotspot computation on apache spark (gis cup). In *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[14] Pradeep Mohan, Ronald E Wilson, Shashi Shekhar, Betsy George, Ned Levine, and Mete Celik. Should sdbms support a join index? a case study from crimestat. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, 2008.

[15] https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu.

[16] J Keith Ord and Arthur Getis. Local spatial autocorrelation statistics: distributional issues and an application. *Geographical analysis*, 27(4):286–306, 1995.

[17] Anmol Paudel and Satish Puri. Openacc based gpu parallelization of plane sweep algorithm for geometric intersection. In *International Workshop on Accelerator Programming Using Directives*, pages 114–135. Springer, 2018.

[18] Anmol Paudel, Jie Yang, and Satish Puri. Parallelization of plane sweep based voronoi construction with compiler directives. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 908–911. IEEE, 2019.

[19] Shangfu Peng, Hong Wei, Hao Li, and Hanan Samet. Simplification and refinement for speedy spatio-temporal hot spot detection using spark (gis cup). In *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[20] Satish Puri, Anmol Paudel, and Sushil K Prasad. MPI-Vector-IO: Parallel I/O and partitioning for geospatial vector data. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–11, 2018.

[21] Satish Puri and Sushil K Prasad. A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using mpi. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 576–585. IEEE, 2015.

[22] Shashi Shekhar, Pusheng Zhang, and Yan Huang. Spatial data mining. In *Data mining and knowledge discovery handbook*, pages 837–854. Springer, 2009.

[23] Scott D. Stoller, Michael Carbin, Sarita Adve, Kunal Agrawal, Guy Blelloch, Dan Stanzione, Katherine Yelick, and Matei Zaharia. Future directions for parallel and distributed computing: Spx 2019 workshop report. *NSF Workshop Reports*, Oct 2019.

[24] Waldo R Tobler. A computer movie simulating urban growth in the detroit region. *Economic geography*, 46(sup1):234–240, 1970.

[25] Jie Yang, Anmol Paudel, and Satish Puri. Spatial data decomposition and load balancing on hpc platforms. PEARC '19, pages 1–4. ACM, Jul 28, 2019.

[26] Jie Yang and Satish Puri. Efficient parallel and adaptive partitioning for load-balancing in spatial join. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 810–820. IEEE, 2020.

[27] Song-lin Zhang and Kun Zhang. Comparison between general moran's index and getis-ord general g of spatial autocorrelation. *Acta Scientiarum Naturalium Universitatis Sunyatseni*, 4:022, 2007.