Spatial Scan Statistics on the GPGPU

Stephen G. Larew, Ross Maciejewski, *Member, IEEE*, Insoo Woo, and David S. Ebert, *Fellow, IEEE*

Abstract— Kulldorff's spatial scan statistic and the software implementation (SaTScan) are widely used for the detection and evaluation of geographic clusters, particularly within the health care community. Unfortunately, the computational time of the scan statistic depends on a wide variety of variables, and, depending on the chosen parameter settings and operations, the computational time can be on the order of seconds to weeks. The greatest factors in computational time are the number of cases in the dataset and the number of time intervals over which these case are being aggregated. Fortunately, the scan statistic algorithm is highly parallelizable, where the runtime can be equally divided over the number of processors used. Given that a Graphics Processing Unit (GPU) is a high performance many-core processor, we can take advantage of the GPU's speed and the parallelizability of the scan statistic algorithm to create a low cost means of efficiently reducing the runtime. In this work, we present an implementation of the spatial scan statistic for the GPU using the CUDA programming language. Our current results focus on purely spatial scan statistics (as opposed to spatiotemporal) with an underlying Bernoulli distribution model. We discuss the resultant speed increase, issues with porting such algorithms to the GPU, modifications to the algorithm for further speed increases, and future ideas for utilizing scan statistics and the GPU in a visual analytics environment.

Index Terms—GPGPU, spatial scan statistics, visual analytics, geovisualization.

1 Introduction

Recently, the detection of adverse health events has focused on prediagnosis information to improve response time. This type of detection is more largely termed *syndromic surveillance* and involves the collection and analysis of statistical health trend data, most notably symptoms reported by individuals seeking care in emergency departments. These are multivariate spatiotemporal datasets, and as such, require complex analytical algorithms for detecting anomalies within the data. However, the confirmation of syndromic anomalies requires expert knowledge to determine potential causes, locations and false positives. As such, our past work [12] focused on developing a visual analytics [19] environment in which users could explore syndromic surveillance data for anomalies. The use of interactive visualizations coupled with underlying statistical methodology for anomaly detection proved to be a useful approach. Unfortunately, many detection methods are computationally infeasible for an interactive environment.

Detecting clusters, or "hotspots," in data has become widespread with applications in fields such as epidemiology, data mining, astronomy, bio-surveillance, forestry, and uranium mining among others. Amongst the many methods for discovering hotspots, the spatial scan statistic [7] developed by Kulldorff has been the most widely adopted. The spatial scan statistic is used for detecting statistically significant clusters of events in space or space-time; however, the calculations of this statistic requires significant amounts of computational power and time. A current implementation of the scan statistic can be found in the SaTScan software package [9].

SaTScan provides a flexible user interface for computing scan statistics for a variety of distributions to detect statistically significant clusters of spatiotemporal events. Spatial cluster detection must take into account several factors such as inhomogeneous population densities, statistical significance, time-varying data, varying cluster size and shape, all properties that are accounted for in the spatial scan statistic. To facilitate such cluster detection, SaTScan utilizes a Monte Carlo [2, 20] simulation to determine the statistical distribution of the scan statistic. A likelihood ratio test is used to determine a p-value for possible clusters in order to reject the null hypothesis. Research has shown that the spatial scan statistic is the *most powerful test* [7] for determining cluster location and existence. Furthermore, the strength of the spatial scan statistic lies in its ability to determine the statistical significance of a possible cluster, thus providing analysts with a certainty metric.

Previously, work has been done on utilizing scan statistics in epidemiological settings. Spatial scan statistics have many applications to healthcare. They have been adapted and used in an early warning system for West Nile virus in syndromic surveillance [14]. As part of daily monitoring of hospital visit records, a few viral outbreaks were detected by the New York City Department of Health and Mental Hygiene using SaTScan [4]. Regional increased rates of prostate cancer were found in the U.S. using the spatial scan statistic, possibly indicating unknown risk factors to be further studied [6]. A study of breast cancer incidence in Massachusetts showed clusters of high incidence even after adjusting for risk factors [18]. Furthermore, work by Neill et al. [15] presented a method for enhancing the speed of detecting spatial temporal clusters by utilizing a novel kd-tree structure.

Unfortunately, current implementations of the scan statistic on commodity PCs are not suitable for the real-time detection of clusters. Fortunately, the algorithms speed may be improved due to its great deal of parallelism since each possible cluster can be scanned independently of all others. By running hundreds or thousands of concurrent threads, one per possible cluster or one per grid point, significant speed boosts can be achieved. While SaTScan does make use of multi-core CPUs, finding clusters in large data sets with a large amount of discrete temporal steps is still non-trivial. In fact, determining the scan statistic and detecting clusters in large data sets can take months to years on high-end desktops. However, the massive parallelism of commodity graphics hardware now provides researchers with the ability to design algorithms that take advantage of GPGPU-based operations [5, 17]. Moreover, the CUDA programming language provides programmers with access to shared memory and fully utilize the massive parallelism of the GPU using a C-like programming language [16]. As such, this work presents an implementation of the spatial scan statistic on the GPGPU.

Such work is an underpinning in visual analytics as interactive data exploration methods that can incorporate end user knowledge is a vital component of the analysis process. The output of spatial scan statistics have been used in a variety of applications (e.g., [1]), and the resulting visualizations of such clusters aids the end user in finding statistically significant regions within their dataset. In this paper, we summarize the statistical theory for spatial scan statistics with a Bernoulli distribution. Next, we describe our modifications to the scan statistic algorithm to enable higher computational performance on the GPU. We

• *S. G. Larew, R. Maciejewski, I. Woo and D.S. Ebert are with the Purdue University Rendering and Perceptualization laboratory, E-mail: {sglarew|rmacieje|iwoo|ebertd}@purdue.edu.*

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

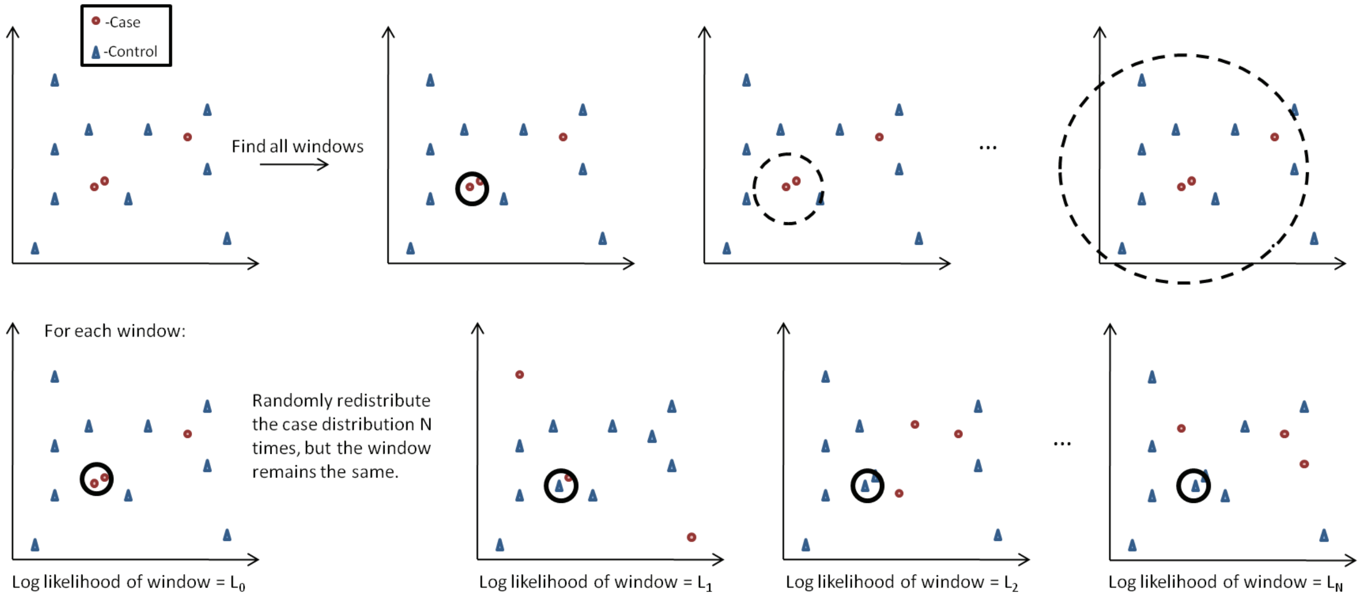

Fig. 1. An illustration of determining window size and the log likelihood calculation steps in the spatial scan statistic. (Top) For each case in the data set, the algorithm iteratively finds all circular windows centered at the case. (Bottom) For each window, a log likelihood value is calculated. The data is redistributed N times and each time, the log likelihood of the window is recalculated.

briefly discuss coding implementation issues (with a full discussion and code samples in the Appendix), and discuss the computational speed up gained by using the GPU. Finally, we discuss future directions for implementations of scan statistics on the GPU and also ideas on utilizing such methods in a visual analytics environment.

2 Spatial Scan Statistics

Kulldorff [7] proposed a *spatial scan statistic* to detect the location and size of the most likely cluster of events in spatial or spatiotemporal data. Using multidimensional data and a varying sized scanning window, the spatial scan statistic will give the location and size of statistically significant clusters of events when compared to a given statistical distribution of events of inhomogeneous density. In this work, we focus only on the spatial scan statistic with a Bernoulli distribution. A detailed description of scan statistics for multiple distributions can be found in [3] and [21] provides details on the scan statistic within the confines of specific applications to health care.

Given a data set in the interval $[a,b]$ bounding a number of randomly placed points, such as illustrated in Figure 1, we can define a scanning window $[t,t+w]$ of size $w<(b-a)$. Figure 1 (Top) shows a number of different scanning windows for the illustrated data set. The scan statistic S_w is the maximum number of points bounded by the scanning window as it slides along the interval. Let $N(A)$ be the number of points in the set A. Then,

$$S_w = \sup_{a<t<(b-w)} N[t,t+w]$$

As the scanning window W of variable size and shape is moved across the 2D area of interest $G \subset V$ for some vector space V, it defines a set of windows $\mathscr{W}$. The probability of an event within a window W is p and the probability of an event outside a window is q. The variable scanning window is illustrated in Figure 1 (Top). Under the null hypothesis, H_0, $p=q$. The alternative hypothesis, H_1 is that there is a window W such that $p>q$.

To determine the probability of events within a window, Kulldorff proposed several models for the underlying data. For our GPU implementation of the spatial scan statistic, we focus only on the Bernoulli distribution model. In this model, each entity in G is in one of two states (either a case or a control as seen in Figure 1). Thus, for any subset $A \subset G$, A follows a binomial distribution. Under the null hypothesis, $N(A) \sim \text{Bin}(\mu(A),p)$. Under the alternative hypothesis, $N(A) \sim \text{Bin}(\mu(A),p)$ for all sets $A \subset W$ and $N(A) \sim \text{Bin}(\mu(A),q)$ for all sets $A \subset W^C$.

For each possible window W in the set of all windows $\mathscr{W}$, a likelihood value $L(W)$ is calculated based on the contents of the window, Figure 1 (Bottom). The likelihood value is maximized over all possible windows and this maximum likelihood is called the scan statistic λ. n_W and n_G are the number of events in a window W and area G respectively. $\mu(G)$ and $\mu(W)$ are total number of points in area G and W respectively.

$$p = \frac{n_W}{\mu(W)}$$

$$q = \frac{n_G - n_W}{\mu(G) - \mu(W)}$$

$$L_0 = \left(\frac{n_G}{\mu(G)}\right)^{n_G} \left(\frac{\mu(G) - n_G}{\mu(G)}\right)^{\mu(G)-n_G}$$

$$L(w) = \begin{cases} p^{n_W}(1-p)^{\mu(W)-n_W} q^{n_G-n_W} \\ \quad (1-q)^{(\mu(G)-\mu(W))-(n_G-n_W)} & \text{if } p>q, \\ L_0 & \text{otherwise} \end{cases}$$

$$\lambda = \max_{w \in \mathscr{W}} \frac{L(w)}{L_0} \tag{1}$$

An analytical description of the distribution of the test statistic does not exist. Thus, a Monte Carlo simulation [2, 20] is used to obtain the distribution of the test statistic under the null hypothesis. As such, for each window, the data is randomly redistributed (Figure 1) and the likelihood value for a given window is calculated under each new distribution. The test statistic for the actual data is then compared to the distribution of the test statistic under the null hypothesis to reject or not reject the null hypothesis with α significance. In other words, once the N likelihood values are calculated, they are sorted in descending order, and the p-value is calculated as the position of λ in the list divided by $N+1$. Finally, when a likelihood value and significance value have been computed for every possible window, a list of scan windows based on location, radius, likelihood and significance are returned.

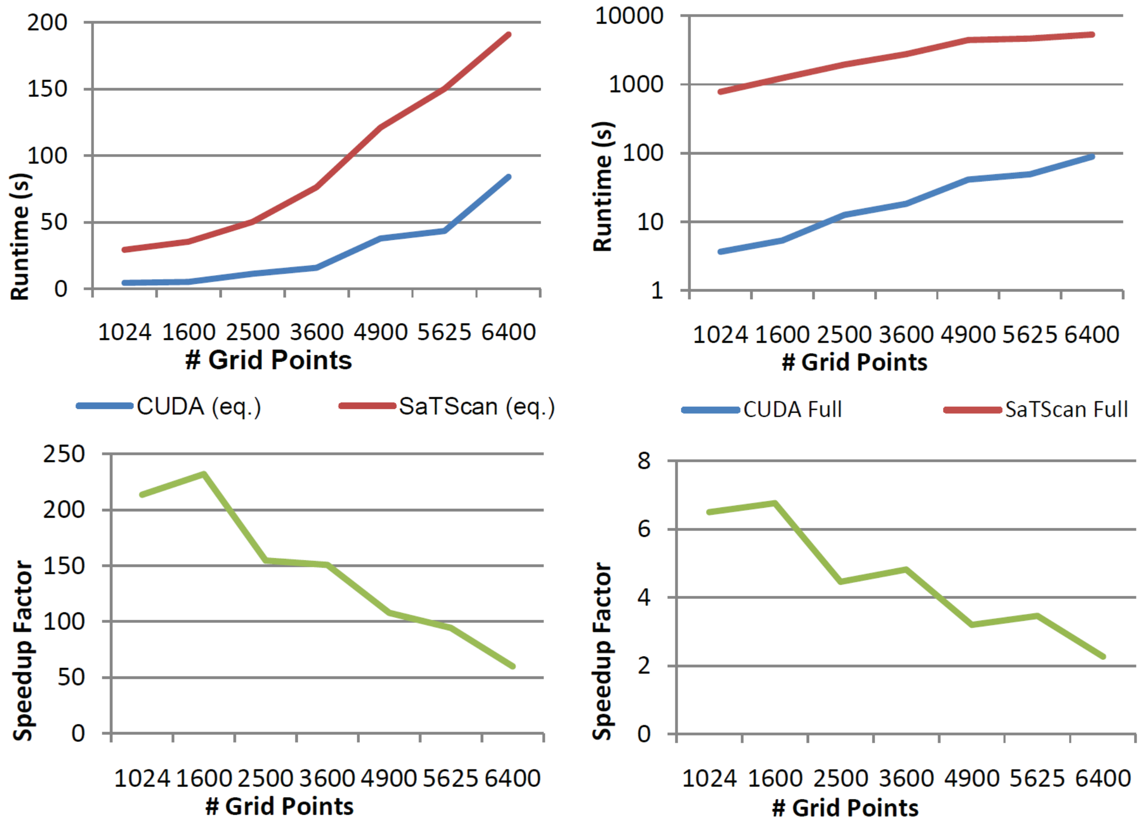

Fig. 2. Timing results of the spatial scan statistic using SaTScan on the CPU versus our implementation on the GPU. (Left)Results with the data gridded to histogram bins input into both programs. (Right)Using the raw data in SaTScan compare with using he histogram binning on the GPU.

3 Implementation

As previously stated, the spatial scan statistic algorithm is highly parallelizable. NVIDIA's CUDA computing model allows a program to run highly concurrent algorithms on a consumer graphics card. CUDA splits computation between a device (GPU) and host (CPU). The host controls the processes run on the device by issuing kernel (function) launches and managing memory transfers. A kernel is a set of threads on the device all running the same function, thus CUDA's execution model is a Single Instruction Multiple Threads (SIMT) model. All CUDA kernels we have developed for the spatial scan statistic can be found in the Appendix.

In order to optimize the performance of a CUDA program, generalized knowledge of the underlying hardware architecture must be utilized to structure CUDA kernels. At the algorithmic level, data parallelism must be exploited. The algorithm should be broken down into small steps that are independent of each other. At the hardware level, memory management and read/write operations are extremely important as well as hardware resource usage. The input to the program is a file containing a list of cases and controls with their grid location, for simplicity, we follow the input structure of the SaTScan software package (see the SaTScan Manual [8] for details).

First, the data is ingested by the host. Next, we noted that only a finite number of unique scanning windows will exist within a given dataset. However, for each case within the dataset, we need to know the list of its nearest neighbors in order to determine the radius of each unique window. In order to simplify the computation and reduce the nearest neighbor calculations, user defined grid is laid out over the data. All data is then aggregated to its nearest grid point. If the grid is very coarse, then the power of the scan statistic is reduced; however, for a fine grid, the results are equivalent to the SaTScan output.

Combining our grid based modification into the spatial scan statistic results in the following algorithm.

Algorithm 1: The spatial scan statistic.

1. For each data point, aggregate to its nearest grid point.

2. For each grid point containing a non-zero number of cases, iteratively increase the window radius. If the new window radius includes grid points that have a non-zero number of cases or controls, save that window centroid and radius.

3. For each window found in step 2, calculate the test statistic according to λ

4. Randomly redistribute the data and repeat step 3 for each Monte Carlo replication to calculate $lambda_i$ saving the list of test statistics for each window.

5. Repeat the redistribution step (step 4) N times.

6. Given the list of N likelihood values for each window, determine the p-value by sorting the list in descending order. The p-value is the position ofλ divided by $N+1$.

Algorithm 1 works for circular windows of variable size, with circle centroids at each grid point. Any discrete, non-homogenous Bernoulli process data set applies. The host machine CPU finds all the windows within the given dataset. The windows are then passed to the GPU, and CUDA kernels, CreateWindows and ScanLocationFull (Found in Appendix A, B and C respectively), are invoked. After each scanlocationfull invocation, a scan is done to find the maximum likelihood. Then significance values and reports are generated.

In implementing this algorithm, we found the greatest speed bottleneck to be in creating the windows. As a window expands in radius to include new grid points, the order in which grid points are added results in random global memory accesses with high latency resulting in extremely low memory throughput. Using texture fetches does

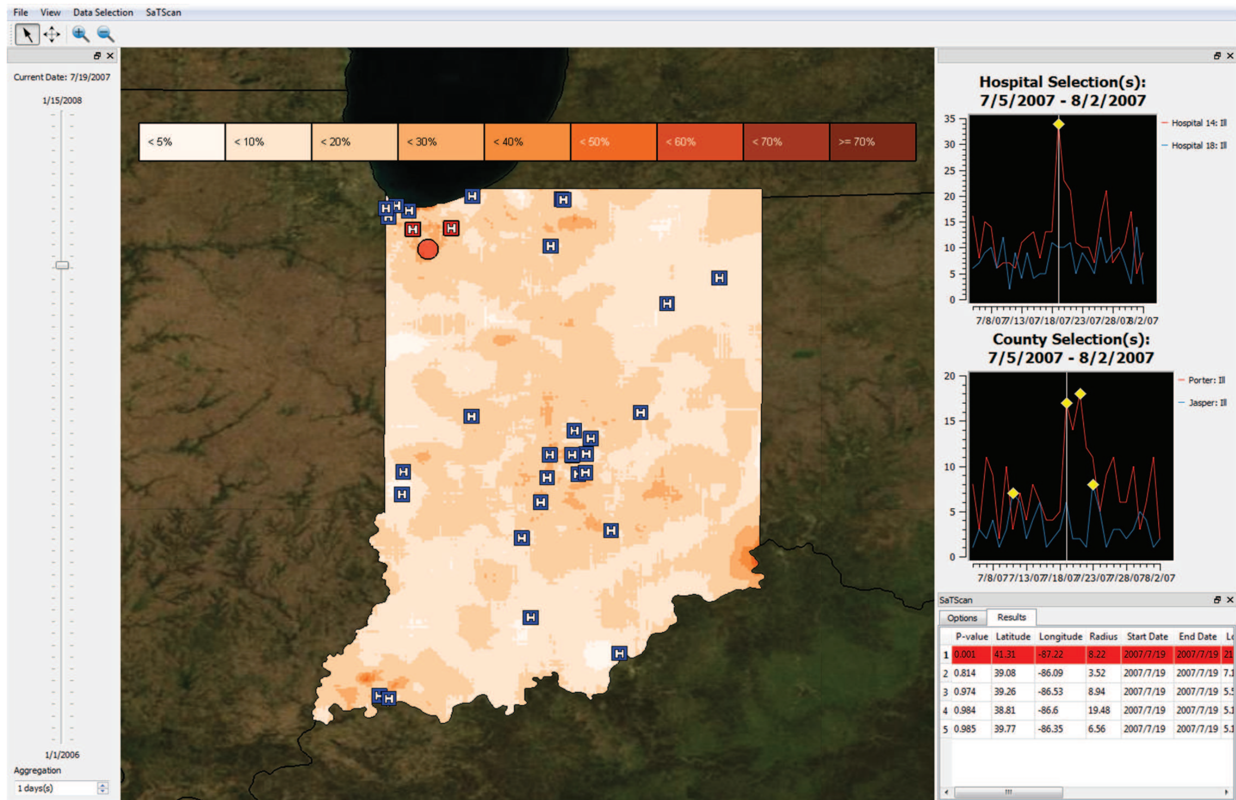

Fig. 3. The visual analytics environment developed by Maciejewski et al. [12] which integrates the SaTScan software into a visual analytics environment.

not result in any speed up and actually makes program run-time more variable. Our solution is to cache grid point data in shared memory. Unfortunately, this solution is limited by the 16 kB of shared memory per multiprocessor. As such, we iteratively load parts of the grid point data into shared memory and work with partial grid point data. This solution is outlined in Algorithm 2 and the implementation is found in the CreateWindows kernel of the Appendix.

Algorithm 2:

1. Load (next) segment i of cases/controls grid data into shared memory.
2. For each nearest neighbor n:
   (a) If n is in the range of i, add the appropriate cases/controls to a window array.
3. Repeat steps 1 and 2 for all segments of grid data.

4 Timing Results

We compared our results to the CPU implementation of the spatial scan statistic using the SaTScan software. All tests were performed on a dual core Intel Xeon 3.2 GHz computer with 2 GB ram and a NVIDIA geForce 8800 gtx 768 MB graphics card.

For verification of our algorithms correctness, we utilized the Northeastern USA Benchmark Data [10] and the New York City Benchmark Data [11]. Output clusters and p-values were compared and verified. Next, we created a synthetic dataset utilizing work from Maciejewski et al. [13] with known clusters. The goal of the synthetic dataset creation was to generate a more realistic syndromic surveillance situation with data streaming from a large number of state emergency departments. Again, results were compared with the output from the SatScan software.

By using CUDA and parallel computing, a modest speedup was obtained when using equivalent input (Figure 2 - Left). Note that the modest speedup is due to the fact that SatScan has already been parallelized in its freely downloadable state. We found the CUDA implementation results in a 2x speed up when compared to the CPU SaTScan software for a data set with 1,000,000 population points. The data set used by both the CUDA implementation and the CPU SaTScan software is aggregated to grid points. Due to the way the CUDA implementation aggregates data to grid points, it gains a significant speed boost over the CPU SaTScan software when non-aggregated data is input to the CPU SaTScan software. Figure 2 (Right) shows a speed boost of over 60 times for the CUDA implementation with aggregated data when compared to the CPU SaTScan software with non-aggregated data. As such, we can see that enhancing the data structures used in spatial scan computations can actually greatly improve the performance of the algorithms.

5 Future Work

In this work, we introduced our approach to using the high parallelism of the GPU for detecting clusters on a single PC with commodity graphics hardware. However, as the sizes of data being collected and analyzed by public health departments is continuously increasing, new means of processing the data also need to be explored. Given that clustering high resolution multivariate and spatial data requires more graphics memory and more computation power for interactive data analysis, we have begun looking at extending our approach to use multi-core CPUs and many-core GPUs through a Message Passing Interface (MPI). However, while recent NVIDIA graphics hardware provides atomic operations and larger graphics memories, the GPU is not an appropriate computational resource for analyzing histogram (grid) based data for large scale datasets, particularly for fine grids resulting in sparse bins. Our plans are to develop a parallel sparse histogram computation for spatial datasets using multicore CPUs and MPI. For better load balancing, we are researching how to optimally store, query and distribute temporal and spatial datasets within these computational resources.

In parallel to these activities, our work has also focused on interactively exploring large health related datasets in a visual analytics environment. Our previous work [12] integrated SaTScan into a comprehensive, predictive visual analytics tool that incorporates several data sources and models for syndromic surveillance and is shown in Figure 3. Another recent visual analytics systems that utilizes the spatial scan statistic is from Chen et al. [1] where the authors look at refining cluster results in a visual analytics environment. As the datasets become larger, a full set of spatiotemporal scan statistics will be unable to run without the use of super computing resources. As such, our future work will focus on having the user interactively define spatiotemporal areas of interest based on more computationally modest statistical algorithms (such as density estimation or temporal control charts). The selected region can then be passed to the scan statistics algorithm, and a p-value can be generated to indicate to the user if a cluster of interest is in fact statistically significant. Furthermore, the user interaction can be refined to draw any arbitrarily shaped window, allowing for a more robust algorithm through the combination of interactive visuals and an underlying statistical methodology.

A COMPLETEGRIDKERNEL

In the CompleteGrid kernel, each thread corresponds to a data point. It iterates over every grid point calculating the distance to that grid point from the data point corresponding to the kernel, keeping track of the nearest grid point. Once complete, the thread adds its corresponding data point's cases and controls data to the nearest grid point. This requires an atomic operation to avoid two threads accessing a grid point's data at once. Therefore, only compute capability 1.1 (see [16] for more information about CUDA compute capability) and above devices will run this kernel.

```
#define COMPLETE_GRID_BLOCK_SIZE 128
#define GRIDPOINTS_LOAD 512

__global__ void CompleteGridKernel(CUDALocations
    Locations, CUDAGridPoints GridPoints)
{
  const size_t LocIdx = blockIdx.x*
      COMPLETE_GRID_BLOCK_SIZE + threadIdx.x;

  __shared__ float2 SubLocs[COMPLETE_GRID_BLOCK_SIZE
      ];
  __shared__ float2 SubPoints[GRIDPOINTS_LOAD];

  if (LocIdx < Locations.num)
  {
    // Load the portion of the Locs array used per
        block into shared memory for better
        performance.
    SubLocs[threadIdx.x] = Locations.pos[LocIdx];

    float ShortestDist = FLT_MAX;
    size_t BestGridPoint;
    size_t NumGridSegments = (GridPoints.num +
        GRIDPOINTS_LOAD - 1)/GRIDPOINTS_LOAD;
    for (size_t i = 0; i < NumGridSegments; ++i)
    {
      // Load segment i into SMEM:

      // Iterate over subsegments of i the size of
          the block.
      for (size_t j = 0; j < GRIDPOINTS_LOAD /
          COMPLETE_GRID_BLOCK_SIZE; ++j)
      {
        // Calculate the index into the grid points
            arrays.
        const size_t idx = i*GRIDPOINTS_LOAD+j*
            COMPLETE_GRID_BLOCK_SIZE + threadIdx.x;
        if (idx < GridPoints.num)
        {
          // Load into SMEM.
          SubPoints[j*COMPLETE_GRID_BLOCK_SIZE +
              threadIdx.x] = GridPoints.pos[idx];
        }
      }

      for (size_t k = 0; k < GRIDPOINTS_LOAD; ++k)
      {
        float w = SubPoints[k].x - SubLocs[threadIdx
            .x].x;
        float h = SubPoints[k].y - SubLocs[threadIdx
            .x].y;
        float dist = w*w+h*h;
        if (dist < ShortestDist) {
          ShortestDist = dist;
          BestGridPoint = i*GRIDPOINTS_LOAD+k;
        }
      }
    }

    // Only works on device of compute capability
        1.1 and above.
    atomicAdd((unsigned int*)&GridPoints.cases[
        BestGridPoint], Locations.cases[LocIdx]);
    atomicAdd((unsigned int*)&GridPoints.controls[
        BestGridPoint], Locations.controls[LocIdx]);
  }
}
```

B CREATEWINDOWS KERNEL

```
#define CREATE_WINDOWS_BLOCK_SIZE 128
#define GRIDPOINTS_LOAD (CREATE_WINDOWS_BLOCK_SIZE
    *15)


__global__ void CreateWindowsKernel(
        CUDAGridPoints GridPoints,
        CUDANeighbors Neighbors,
        UDAWindows Windows)
{
  // idx is which grid point we're scanning from.
  const size_t GridIdx = blockIdx.x*
      CREATE_WINDOWS_BLOCK_SIZE + threadIdx.x;

  __shared__ size_t GridCases[GRIDPOINTS_LOAD];
  __shared__ size_t GridControls[GRIDPOINTS_LOAD];

  // For each segment i of global cases/controls
      memory:
  size_t NumGridSegments = (GridPoints.num +
      GRIDPOINTS_LOAD - 1)/GRIDPOINTS_LOAD;
  for (size_t i = 0; i < NumGridSegments; ++i)
    // Load segment i into SMEM:

    // Iterate over subsegments of i the size of the
        block.
    for (size_t j = 0; j < GRIDPOINTS_LOAD /
        CREATE_WINDOWS_BLOCK_SIZE; ++j)
    {
      // Calculate the index into the grid points
          arrays.
      const size_t idx = i*GRIDPOINTS_LOAD+j*
          CREATE_WINDOWS_BLOCK_SIZE + threadIdx.x;
      if (idx < GridPoints.num)
      {
        // Load into SMEM.
        GridCases[j*CREATE_WINDOWS_BLOCK_SIZE +
            threadIdx.x] = GridPoints.cases[idx];
        GridControls[j*CREATE_WINDOWS_BLOCK_SIZE +
            threadIdx.x] = GridPoints.controls[idx];
      }
```

```
    }

    __syncthreads();

    size_t* WindowsCases = Windows.cases + GridIdx;
    size_t* WindowsControls = Windows.controls +
        GridIdx;
    size_t* NeighborIdx = Neighbors.idx + GridIdx;

    // For each nearest neighbor n:
    for (size_t j = 0; j < GridPoints.num; ++j)
    {
      // If n is in the range of i:
      size_t n = *NeighborIdx - i*GRIDPOINTS_LOAD;
      if (n < GRIDPOINTS_LOAD)
      {
        *WindowsCases = GridCases[n];
        *WindowsControls = GridControls[n];
      }

      WindowsCases = (size_t*)((char*)WindowsCases +
          Windows.casesPitch);
      WindowsControls = (size_t*)((char*)
          WindowsControls + Windows.controlsPitch);
      NeighborIdx = (size_t*)((char*)NeighborIdx +
          Neighbors.idxPitch);
    }
    __syncthreads();
  }
}
```

C SCANLOCATIONFULL KERNEL

There are two variants of the ScanLocation kernel. ScanLocationFull kernel saves all relevant data about each cluster and is listed here. ScanLocationNull is used on the Monte Carlo replications and only saves log likelihood values. Both kernels iterate over each window in the windows array generated by the CreateWindows kernel and calculate log likelihood values.

```
#define SPATIAL_SCAN_BLOCK_SIZE 128

__global__ void ScanLocationFullKernel(
size_t NumGridPoints,
float2* GridPointsPos,
float* Radii,
size_t RadiiPitch,
CUDAWindows Windows,
CUDAClusters Clusters,
float MaxWindowSize,
size_t TotalPopulation,
size_t TotalCases)
{
  // idx is which grid point we're scanning from.
  const size_t GridIdx = blockIdx.x*
      SPATIAL_SCAN_BLOCK_SIZE + threadIdx.x;

  if (GridIdx < NumGridPoints)
  {
    float BestLikelihood = 0.0f;
    float BestRadius = -1.0f;
    float BestPopulation = 999999, BestCases =
        999999;

    size_t NumCasesWindow = 0, NumControlsWindow =
        0;
    size_t* WindowsCases = Windows.cases + GridIdx;
    size_t* WindowsControls = Windows.controls +
        GridIdx;
    float* Radius = Radii + GridIdx;

    for (size_t i = 0; i < NumGridPoints; ++i)
    {
      // Update the window and outside stats.
      NumCasesWindow += *WindowsCases;
      NumControlsWindow += *WindowsControls;
      size_t TotalWindow = NumCasesWindow +
          NumControlsWindow;
      size_t NumCasesOutside = TotalCases -
          NumCasesWindow;
      size_t TotalOutside = TotalPopulation -
          TotalWindow;
      size_t NumControlsOutside = TotalOutside -
          NumCasesOutside;

      // Only run scan for windows that include less
          than MaxWindowSize % of population
      if (TotalWindow > (size_t)(MaxWindowSize*
          TotalPopulation))
        break;

      // Calculate the likelihood for the current
          window.
      // Refer to eqs. 14.2 and 14.3 (Kulldorf,
          1999)
      if ((float)NumCasesWindow/TotalWindow > (float
          )NumCasesOutside/TotalOutside)
      {
        // Calculate the log of the likelihood ratio
            .
        float Likelihood = ((NumCasesWindow!=0)?((
            float)NumCasesWindow * logf((float)
            NumCasesWindow/TotalWindow)):0.0f) +
        ((NumControlsWindow!=0)?((float)
            NumControlsWindow * logf((float)
            NumControlsWindow/TotalWindow)):0.0f) +
        ((NumCasesOutside!=0)?((float)
            NumCasesOutside * logf((float)
            NumCasesOutside/TotalOutside)):0.0f) +
        ((NumControlsOutside!=0)?((float)
            NumControlsOutside * logf((float)
            NumControlsOutside/TotalOutside)):0.0f)
            -
        ((TotalCases!=0)?((float)TotalCases * logf((
            float)TotalCases/TotalPopulation)):0.0f)
             -
        (((TotalPopulation - TotalCases)!=0)?((float
            )(TotalPopulation - TotalCases) *
        logf((float)(TotalPopulation - TotalCases)/
            TotalPopulation)):0.0f);

        // If the current window is more likely,
            remember this.
        if (Likelihood > BestLikelihood)
        {
          BestLikelihood = Likelihood;
          BestRadius = *Radius;
          BestPopulation = TotalWindow;
          BestCases = NumCasesWindow;
        }
      }

      // Increment the neighborIdx and neighborDist
          to the next nearest (closest distance)
          location.
      WindowsCases = (size_t*)((char*)WindowsCases +
          Windows.casesPitch);
      WindowsControls = (size_t*)((char*)
          WindowsControls + Windows.controlsPitch);
      Radius = (float*)((char*)Radius + RadiiPitch);
    }

    float expected = (float)BestPopulation * (float)
        TotalCases / (float)TotalPopulation;
    Clusters.pos[GridIdx] = GridPointsPos[GridIdx];
    Clusters.radius[GridIdx] = BestRadius;
```

```
        Clusters.likelihood[GridIdx] = BestLikelihood;
        Clusters.population[GridIdx] = BestPopulation;
        Clusters.cases[GridIdx] = BestCases;
        Clusters.expectedCases[GridIdx] = expected;
        Clusters.relativeRisk[GridIdx] = ((float)
            BestCases / expected) / ((float)(TotalCases-
            BestCases) / (TotalCases-expected));
    }
}
```


ACKNOWLEDGMENTS

This work is supported by the U.S. Department of Homeland Security's VACCINE Center under Award Number 2009-ST-061-CI0001 and by the US National Science Foundation under Grants OCI-0906379.

REFERENCES

[1] J. Chen, R. E. Roth, A. T. Naito, E. J. Lengerich, and A. M. MacEachren. Geovisual analytics to enhance spatial scan statistic interpretation: An analysis of u.s. cervical cancer mortality. *International Journal of Health Geographics*, 7:57, 2008.

[2] M. Dwass. Modified randomization tests for nonparametric hypotheses. *The Annals of Mathematical Statistics*, 28(1):181–187, 1957.

[3] J. Glaz, J. Naus, and S. Wallenstein. *Scan Statistics*. Springer-Verlag, New York, 2001.

[4] R. Heffernan, F. Mostashari, D. Das, A. Karpati, M. Kulldorff, and D. Weiss. Syndromic surveillance in public health practice, new york city. *Emerging Infectious Diseases*, 10(5), May 2004.

[5] D. Horn. Stream reduction operations for gpgpu applications. In H. Nguyen, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 573–583. Addison Wesley, 2007.

[6] A. Jemal, M. Kulldorff, S. S. Devesa, R. B. Hayes, and J. Joseph F. Fraumeni. A geographic analysis of prostate cancer mortality in the united states, 1970-89. *International Journal of Cancer*, 101:168–174, 2002.

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

[8] M. Kulldorff. Satscan user guide for version 8.0, 2009.

[9] M. Kulldorff and I. Information Management Services. Satscan v8.0: Software for the spatial and space-time scan statistics, 2009.

[10] M. Kulldorff, T. Tango, and P. J. Park. Power comparisons for disease clustering tests. *Computational Statistics adn Data Analysis*, 42:665–684, 2003.

[11] M. Kulldorff, Z. Zhang, J. Hartman, R. Heffernan, L. Huang, and F. Mostashari. Evaluating disease outbreak detection methods: Benchmark data and power calculations. *Morbidity and Mortality Weekly Report*, 53:144–151, 2004.

[12] R. Maciejewski, R. Hafen, S. Rudolph, S. G. Larew, M. A. Mitchell, W. S. Cleveland, and D. S. Ebert. Forecasting hotspots - a predictive analytics approach. *IEEE Transactions on Visualization and Computer Graphics*, To appear.

[13] R. Maciejewski, R. Hafen, S. Rudolph, G. Tebbetts, W. S. Cleveland, S. J. Grannis, and D. S. Ebert. Generating synthetic syndromic surveillance data for evaluating visual analytics techniques. *IEEE Computer Graphics & Applications*, 29:18–28, May/June 2009.

[14] F. Mostashari, M. Kulldorff, J. Hartman, J. Miller, and V. Kulasekera. Dead bird clusters as an early warning system for west nile virus activity. *Emerging Infectious Diseases*, 9:641–646, 2003.

[15] D. B. Neill, A. Moore, and M. Sabhnani. Detecting elongated disease clusters. *Morbidity and Mortality Weekly Report*, 54:197, 2005.

[16] NVIDIA Corporation. Nvidia cuda compute unified device architecture, Nov 2007.

[17] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.

[18] T. J. Sheehan, L. M. DeChello, M. Kulldorff, D. I. Gregorio, S. Gershman, and M. Mroszczyk. The geographic distribution of breast cancer incidence in massachusetts 1988 to 1997, adjusted for covariates. *International Journal of Health Geographics*, 3, 2004.

[19] J. J. Thomas and K. A. Cook, editors. *Illuminating the Path: The R&D Agenda for Visual Analytics*. IEEE Press, 2005.

[20] B. W. Turnbull, E. J. Iwano, W. S. Burnett, H. L. Howe, and L. C. Clark. Monitoring for clusters of disease: Application to leukemia incidence in upstate new york. *American Journal of Epidemiology*, 132(supp1):136–143, 1990.

[21] L. A. Waller and C. A. Gotway. *Applied spatial statistics for public health data*. Wiley, Hoboken, NJ, 2004.