Accelerated Discovery of Discrete M-Clusters/Outliers on the Raster Plane Using Graphical Processing Units

Christian Trefftz¹, Joseph Szakas², Igor Majdandzic¹, and Gregory Wolffe¹

¹ School of Computing, Grand Valley State University, Allendale, MI 49401

² Computer Information Systems, Univ. of Maine-Augusta, Augusta, ME 04330
{trefftzc,wolffe}@gvsu.edu, majdanig@student.gvsu.edu,
szakas@maine.edu

Abstract. This paper presents two discrete computational geometry algorithms designed for execution on Graphics Processing Units (GPUs). The algorithms are parallelized versions of sequential algorithms intended for application in geographical data mining. The first algorithm finds clusters of m points, called m -clusters, in the rasterized plane. The second algorithm complements the first by identifying outliers, those points which are not members of any m -clusters. The use of a raster representation of coordinates provides an ideal data stream environment for efficient GPU utilization. The parallel algorithms have low memory demands, and require only a limited amount of inter-process communication. Initial performance analysis indicates the algorithms are scalable, both in problem size and in the number of seeds, and significantly outperform commercial implementations.

Keywords: GPU algorithms, Geographical data mining, CUDA programming.

1 Introduction

Traditional computational geometry algorithms are applied to subsets of the plane, using real values to designate point coordinates. Thus, there are an infinite number of points in the domain of the problem.

In Geographical Information Systems (GIS) the use of a raster representation defines a finite number of points (or pixels) in the problem domain. There are several advantages to using a raster of vector data for spatial processing: the raster data model is well-suited for GIS applications aimed at the analysis of the spatial variability of geographic phenomena, it is more effective in the representation of continuous surfaces, and it can be exploited for accelerated processing and display [1].

Recently, Gudmundsson, van Kreveld and Narasimhan [2] presented several computational geometry algorithms suitable for applications in geographical data mining. Their algorithms find clusters of points on the plane and identify outliers, assuming two types of land cover and defining a cluster via its bounding square. In this paper, we present parallelized algorithms to solve the same problems without these assumptions and restrictions. Furthermore, our algorithms are optimized to execute on Graphics Processing Units (GPUs). GPUs, in addition to being highly parallelized computational units, are particularly well-suited for spatial applications.

1.1 Graphics Processing Units

GPUs, ubiquitous on video cards and familiar to the gaming community, have been steadily increasing in speed, capacity and number of processing units. Given the insatiable demand of modern graphics applications, it seems reasonable to assume that they will continue to do so. Until recently, GPU usage was largely limited to the rendering expected of a video card. That has begun to change with the advent of General Purpose computation on Graphics Processing Units (GPGPU), and the new languages and environments developed in support of that goal. Languages used to program GPUs typically use the Single Instruction Multiple Data (SIMD) style of coding. A program specifies a sequence of operations that is to be applied to the vertices of a polygon or to the pixels comprising an image to be rendered. These operations execute in parallel on different processing units, each on their own unique vertex or pixel.

General purpose computation on GPUs extends this methodology to operate on non-rendering data. For example, one of the new GPU programming languages is called CUDA (Computer Unified Device Architecture) [3], designed for NVIDIA's GPUs. CUDA has a number of elements in common with other Data Parallel Languages, such as MPL [4], C* [5] and Parallaxis [6]. In CUDA, the programmer specifies a *kernel*, a piece of code that is executed in parallel on all of the processing elements in a GPU. During parallel execution, each processing element is responsible for one pixel (or data object), and the kernel contains precisely the set of operations that is to be applied to every element. Programmer-defined "blocksize" and "gridsize" attributes partition the data and direct its redistribution to processing elements.

CUDA's run time environment schedules the parallel execution of the code in the kernel on the multiple processing units inside the GPU. The local variables within a kernel belong to a particular instance of a kernel and each instance of the kernel has its own version (copy) of those variables. In some ways the kernel resembles a class; many instances of the kernel may be active at the same time, each with its own "instance" local variables.

Special operations allow the programmer to move data between main memory on the "host" computer and the device memory on the GPU card.

1.2 Computational Geometry / Geographical Data Mining

Designing and developing algorithms for GPU execution is an active research area. Some of these algorithms use graphic primitives to find approximate solutions to computational geometry problems. Given the nature of GPUs and the displays upon which solutions are ultimately rendered, it is natural to work in the domain of pixels, using integer coordinates. For example, Hoff et al. [7] describe an algorithm to calculate generalized Voronoi diagrams. Fan et al. [8] based their algorithm for natural neighbor interpolation on the same discretization of the plane. Denny [9,10] uses GPU primitives to implement algorithms that solve the problems of: determining the smallest homothetic scaling of a star-shaped polygon enclosing a set of points, finding the largest empty homothetic scaling of a polygon completely contained inside an arbitrary region, and adding a new point to an existing Voronoi diagram such that the resulting Voronoi region has maximal area.

The remainder of this paper is structured as follows: a parallelized algorithm for finding clusters of m points is described in section 2. Section 3 contains a modified algorithm for finding outliers. Initial performance results are presented in section 4. Finally, section 5 presents our conclusions.

2 Finding M-Clusters

Clusters can be defined in different ways [11]. One way is to think of a cluster as a "large enough" subset of points that are close to each other. In GIS, finding properties of point sets is known as point pattern analysis [12].

Based on this notion of a cluster, we present the following definitions: let P be a set of n points in the plane, representing the objects that are being analyzed for clustering. These n points are known as seeds. Let r denote a distance of interest. The set of points in a cluster should lie inside a circle of radius r or smaller. Let m denote the minimum number of points required to form a cluster. The problem then, is to identify subsets of m points (or more) that are inside a circle of radius r or smaller. We refer to this group of points as an m -cluster.

Gudmundsson's paper contains a detailed discussion of the issue of the precision of the results when one "snaps" the coordinates of the seeds to integer coordinates, and one of the algorithms presented in that paper uses integer coordinates for the seeds. The following example illustrates the integer approach.

Consider a raster surface of size 12×12 with 4 seeds located at the coordinates: (4,4), (5,10), (6,2) and (7,9). Assume that we have specified that we need $m = 3$ seeds per cluster and that the radius of every m -cluster should be $r = 4.2$ or smaller. For this particular set of input data, two points qualify as centers of 3-clusters: (6,5) and (6,6). In the raster surface of Figure 1, the 'x' symbol denotes the locations of the seeds and '.' denotes the locations of the centers of m -clusters.

Fig. 1. An example showing two 3-clusters, one centered at (6,5) and another at (6,6)

Note that a seed may be a member of more than one m -cluster, and that an m -cluster may contain more than m seeds. Note also that more than one pixel may be a candidate center for the same m -cluster. The criterion for choosing a particular center

for an m -cluster depends on the specific application. For example, it may be important to choose the candidate center pixel with the smallest resultant m -cluster radius.

2.1 Sequential M-Clustering Algorithm

We next describe a sequential algorithm that calculates the set of m -clusters for a set P of points. The area is represented by a raster surface of size $xSize \times ySize$, which is the sub-plane of interest. The seeds are assumed to be inside the area covered by the raster surface. The algorithm looks for points on the raster surface that are the (approximate) centers of circles of radius r or smaller that contain at least m points, out of the original n points. The pseudo-code for this algorithm is given in Figure 2.

```

For every pixel  $x$  in the raster surface
  Initialize counter to 0
  For every seed  $s$  in the set of  $n$  seeds
    Find the distance  $d$  from pixel  $x$  to seed  $s$ 
    If distance  $d$  is less than or equal to radius  $r$ 
      Increment counter by 1
  If counter is greater than or equal to  $m$  then
    This pixel on the raster surface is the center
    of a circle of radius  $r$  or smaller that
    contains at least  $m$  out of the  $n$  seeds, thus
    this is the center of an  $m$ -cluster

```

Fig. 2. Pseudo-code to find m -clusters on a raster surface

The complexity of this sequential algorithm is $O(xSize * ySize * n)$. Gudmundsson's algorithm uses bounding squares instead of circles to define the distance (radius) test; and it is not equivalent to compare complexities with the algorithm presented here. It is clear that the complexity of the presented algorithm is high; however, it is not intended to be competitive with other approaches in a sequential setting. Its advantage lies in its suitability for highly efficient parallelism on a GPU.

2.2 Parallelized M-Clustering Algorithm

The pseudo-code presented in Figure 2 correctly calculates an approximation to the m -clusters but is inefficient when only a single processor is available. However, during the early 1990s, massively parallel machines such as the MasPar and the Connection Machine 2 allowed programmers to devise algorithms based on the assumption that a processor was available for every pixel [13]. As the processing power and the number of processing units inside a GPU increases, these types of algorithms are becoming relevant again. Fundamentally, a GPU's multiple processing units work in tandem in the same SIMD fashion as the earlier generation of parallel machines.

Flynn wrote: "If parallel processors are going to solve ... problems, the problems' representation should be designed for these machines. ... We need to represent problems in a cellular form, which could be addressed by computers in which each element has its own memory. This is crucial if we want to effectively use parallel processors." [14]

The sequential algorithm given in Figure 2 lends itself nicely to a rewrite in a programming language for GPUs, precisely in the spirit of Flynn's statement. Each processing element, which will be responsible for calculations of one pixel, need only contain enough local memory to hold the locations of all of the seeds. An added benefit is that the parallel version of the algorithm works with limited interprocess communications, as all calculations are local.

Since each processing element is responsible for one pixel, the computations performed in the sequential code for every element of the raster surface compose the CUDA kernel. The following pseudo-code sketches the CUDA implementation of the sequential algorithm presented above in Figure 2.

```
// In the Host
Allocate seeds in constant memory
Copy seeds to GPU
Allocate space (matrix) in GPU global memory
Clear matrix
Launch the kernel
Synchronize threads
Copy matrix back to Host
Free memory

// In the GPU (one pixel per processing element):
NumOfClosePoints = 0;
For each seed s
Calculate the distance from this pixel to seed s
    If (distance <= radius)
        NumOfClosePoints++;
If (NumOfClosePoints >= m)
Mark this pixel as the center of an m-cluster
```

Fig. 3. CUDA pseudo-code to find m-clusters on the raster surface

Notice that there are no data dependencies among instances of the kernel, implying that each instance of the kernel can execute independently of all others. The amount of memory declared and used inside each kernel is extremely small, which is important when programming today's GPUs with their limited number of internal registers.

Processing units inside GPUs are called stream processors. Current high-end GPUs contain 128 stream processors. CUDA's run-time environment manages

scheduling and execution, in parallel, of as many instances of the kernel as the number of available stream processors. Thus, if there is one stream processor available for every pixel in the raster surface, then all kernels can execute simultaneously on the entire data set. Therefore, the time complexity of this parallel approximate algorithm is $O(n)$ where n is the number of seeds.

If the number of required instances of the kernel is larger than the number of processing units, i.e. the number of pixels in the problem is greater than the number of processing elements, then the run-time environment will schedule "batches" of kernel executions. This process continues until all required kernel instances have finished their execution.

Thus the parallel algorithm is independent of the number of available stream processors. If the number of pixels exceeds the number of stream processors, computation is automatically batched. If the number of available stream processors increases, the algorithm scales appropriately and correctly.

This algorithm is similar to a parallel algorithm for finding the Voronoi diagram and another for finding the Order- k Voronoi diagram as presented in [15]. The Voronoi diagram algorithm has recently been implemented using CUDA on an NVIDIA GeForce GPU and the complexity of the parallel algorithm has been shown to be linear on the number of seeds [16].

3 Finding Outliers

Statistically, an outlier is a point that is distant from the rest of the data and therefore does not belong. Using the definitions presented in section 2, we define an outlier as a point (a seed) that does not belong to any of the m -clusters found in the original set of points P . Correspondingly, it is possible to modify the cluster algorithm presented in the previous section to instead identify outliers.

The key observation is that as the algorithm calculates the center-to-seed distances required to identify an m -cluster, it is possible to record the identifications of the seeds that correspond to those distances. Thus, once a point in the raster surface is verified as the center of a circle of radius r or smaller that contains at least m seeds, the identities of each of those m seeds in the m -cluster are known. By definition, any seed not belonging to an m -cluster is an outlier.

The following example illustrates a set of seeds with one outlier. The raster surface is again assumed to be of size 12×12 . There are five seeds with coordinates (1,1), (8,8), (8,10), (9,8) and (9,10). For this example, the maximum cluster radius r is defined to be 2.0 and each cluster must contain at least $m = 3$ seeds.

The state of the raster surface is depicted in Figure 4. In this diagram, the locations of the seeds are represented by the '+' symbol. The outlier seed, represented with an '*', is the seed located at coordinate (1,1).

Since our outlier algorithm is based on a modification of the m -clustering algorithm given in the previous section, we do not present any pseudo-code. A description of the required modifications follows.

In the sequential version of the outlier code, an array of counters, containing one entry for every seed, is initialized to zeroes. Whenever a point is identified as the

Fig. 4. The seed located at (1,1) is an outlier

center of an m -cluster, the counters for each of the seeds belonging to that m -cluster are incremented. Once the entire raster surface has been examined, the counters for the outlier seeds will still contain zeroes.

In the parallelized version of this code, each kernel maintains an array of integers called `localInCluster`. The array `localInCluster` contains one entry for every seed and all of entries are initialized to zero. A second array, referred to as `globalInCluster`, is maintained in the shared memory area of the GPU. It also contains an entry for each seed, again initialized to zero. Each time the distance between a seed and the pixel associated with a kernel is determined to be within the radius r , the corresponding entry in `localInCluster` is set to 1. Thus, after calculating the distances from this pixel to all seeds, `localInCluster` contains a zero only in those entries corresponding to seeds that are at a distance greater than the radius from this pixel. If it is determined that the pixel associated with this kernel is the center of an m -cluster, all non-zero values in `localInCluster` are written to their corresponding entries in `globalInCluster`. When all kernels have completed execution, the entries in `globalInCluster` will contain zeroes only in the entries corresponding to seeds that were not part of any cluster. Those seeds are precisely the outliers.

The complexity of the parallel algorithm is again determined to be $O(n)$, where n is the number of seeds. Interestingly, the execution time of this algorithm was found to be very similar to that obtained for the m -clustering code. Unlike the data independent nature of the m -clustering code, the outlier code manifests the possibility of “write” contentions when updating the entries in the `globalInCluster` array. However, the similar execution times and speedups we obtained seem to indicate that contention is not a significant problem.

4 Results

To test the correctness of the parallel SIMD algorithms and the efficiency of the GPU implementations, the algorithms were executed on a machine with an NVIDIA GTS 8800 video card containing 128 stream processors.

There is no publicly available implementation of the Gudmundsson algorithm. However, there are commercial GIS programs that solve the same problem, which we use for comparison, to calculate the speedups obtained by our GPU implementation.

Several experiments were performed to test the scalability of our algorithms. Table 1 contains the execution times of the m-clustering algorithm for a set of 1024 seeds with increasing problem size, i.e. number of pixels. A superset of the data given in Table 1 is plotted in Figure 5.

Table 1. Execution time of the m-clustering algorithm for increasing problem size. Number of seeds is fixed at 1024.

Image size	Number of pixels	Execution time (ms)
256	65536	16.39
512	262144	43.35
1024	1048576	135.37
2048	4194304	473.14
4096	16777216	1703.89

Fig. 5. Scalability of the m-clustering algorithm

The table and corresponding graph demonstrate that the algorithm scales well as the size of the image increases.

The next set of experiments tested whether the algorithm could scale linearly on the number of seeds. The size of the raster surface was fixed at 2048 × 2048, meaning the problem size was fixed at 4194304 pixels. As in the above experiment, the GTS 8800 video card was the test platform. Executions times are given below in Table 2 and plotted in Figure 6.

Once again it is apparent that the algorithm performs efficiently, scaling linearly as the number of seeds is increased. This suggests that this combination of data parallel algorithm and GPU technology will continue to be effective on ever larger datasets.

Table 2. Execution time of the m-clustering algorithm for increasing number of seeds. Problem size (number of pixels) is fixed at 4194304.

Number of seeds	Execution time (ms)
256	227.87
512	317.01
768	407.50
1024	498.71
1280	586.99
1536	677.40
1792	766.81
2048	856.44

Fig. 6. Scalability of the m-clustering algorithm

Speedup tests were conducted against the commercial program ArcGIS v9.2 running on an Intel Core 2 duo 2.16GHz processor with 2GB RAM. The tests involved performing cluster and outlier analysis on the US Dam dataset. Given 1000 points (seeds), ArcGIS took 17 sec., while our algorithm took 0.499 sec. for 1024 points, a speedup over 34. To analyze 2000 points, ArcGIS took 36 sec., while our approach took 0.856 for 2048 points, a speedup over 42. Clearly, the sub-second performance of the parallelized method enables a more “interactive” or real-time experience for users. In addition, the large speedups suggest that it will be possible to perform data mining on the much larger datasets expected of the future.

5 Conclusion

We have presented two parallel raster algorithms optimized for GPUs: the first one finds clusters of at least m points in size, and the second one discovers outliers. Both algorithms have been implemented using CUDA, a programming language for GPUs. The performance results are encouraging and indicate that the algorithms are fast, scalable, linear in the number of seeds, and able to significantly outperform the implementation found in commercial programs.

The algorithms presented here take advantage of large numbers of stream processors and should scale well as the number of stream processors increases in newer GPUs. The low memory demand and the use of a raster format make these algorithms highly efficient in GIS spatial processing. Today's GPUs promise significant performance improvement for applications using raster processing in GIS systems.

References

1. Lo, C.P., Yeung, A.K.W.: Concepts and Techniques in Geographic Information Systems. Pearson Prentice Hall (2007)
2. Gudmundsson, J., van Kreveld, M., Narasimhan, G.: Region-Restricted Clustering for Geographic Data Mining. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 399–410. Springer, Heidelberg (2006)
3. Buck, I.: GPU computing with NVIDIA CUDA. In: Proceedings of 34th Conference on Computer Graphics (SIGGRAPH). ACM, New York (2007)
4. Christy, P.: Software to support massively parallel computing on the MasPar MP-1. In: Proceedings 35th IEEE Computer Society Conference (Compcon). IEEE, Los Alamitos (1990)
5. Rose, J.R., Steele, G.I.: C*: An extended C language for data parallel programming. In: Proceedings 2nd International Conference on Supercomputing (SC), pp. 2–16. ACM/IEEE (1988)
6. Braunl, T.: Parallax-III: Architecture-Independent Data Parallel Processing. Transactions on Software Engineering 26, 227–243 (2000)
7. Hoff, K.E., Culver, T., Keyser, J., Lin, M., Manocha, D.: Fast Computation of Generalized Voronoi Diagram and the Order-k Voronoi Diagram. In: Proceedings of 26th Conference on Computer Graphics (SIGGRAPH). ACM, New York (1999)
8. Fan, Q., Efrat, A., Koltun, V., Krishnan, S., Venkatasubramanian, S.: Hardware-Assisted Natural Neighbor Interpolation. In: Proceedings of 7th Workshop on Algorithm Engineering and Experimentns (ALENEX). SIAM, Philadelphia (2005)
9. Denny, M.: Algorithmic Geometry via Graphics Hardware. PhD thesis, Universitaet des Saarlandes (2003)
10. Denny, M.: Solving Geometric Optimization Problems using Graphics Hardware. Computer Graphics Forum 22, 441–451 (2003)
11. Jain, A.K., Murty, M.N., Flynn, P.J.: Data Clustering: A Review. ACM Computing Surveys 31, 264–323 (1999)
12. O'Sullivan, D., Unwin, D.J.: Geographic Information Analysis. John Wiley and Sons, Chichester (2003)
13. Blueloch, G.E., Little, J.J.: Parallel Solutions to Geometric Problems on the Scan Model of Computation. Tech. Report A.I. Memo 952, MIT Artificial Intelligence Lab (1988)
14. Flynn, M.J.: Parallel processors were the future...and yet be. Computer 29, 151–152 (1996)
15. Trefftz, C., Szakas, J.: Parallel Algorithms to find the Voronoi Diagram and the Order-k Voronoi Diagram. In: Proceedings 3rd Workshop on Massively Parallel Processing (IPDPS). IEEE, Los Alamitos (2003)
16. Majdandzic, I., Trefftz, C., Wolffe, G.: Computation of Voronoi Diagrams using a Graphics Processing Unit. In: Proceedings of 2008 IEEE Intl. Conf. on Electro/Information Technology (EIT). IEEE, Los Alamitos (2008)