Locating Critical Points in 3D Vector Fields using Graphics Hardware

David Hyon Berrios [*]
University of Maryland, Baltimore County

Abstract

Calculating critical points in 3D vector fields is computationally expensive, and requires parameter adjustments to maximize the number and type of critical points found in a particular 3D vector field. This paper details the modifications required to execute Greene's bisection method directly on graphics hardware, and compares the performance of the CPU+GPU versions to the CPU only versions.

Keywords: critical points, singularities, vector fields, null points, GPGPU

1 Introduction

3D vector fields are used in computational fluid dynamics, weather, and physics simulations, among others. Because the size of the simulations usually grows as the computational capabilities increase, better methods of reducing the search space within the simulation bounds becomes more important. Manual methods of searching for features can no longer keep pace with the increase in the size of the datasets. Similarly, visualizing the evolution of structural changes within the 3D vector fields through time becomes more difficult as the 3D vector fields become larger (higher resolution, both temporal and spatial) and the simulations become longer (more timesteps).

An example can illustrate the difficulty of searching a 3D vector field for a particular feature. Suppose a user wanted to locate a tornado in a weather simulation that spanned the entire earth. Normally, thresholds could be used to locate regions where the wind speed was higher than the threshold. Although the wind speed is a good heuristic for locating tornadoes, it does not uniquely identify their locations. Hurricanes can also have strong winds, but their behaviors and characteristics are vastly different. Alternatively, the effort of finding this tornado in both the space of each individual timestep, and across multiple timesteps, can be offset by using critical points. A tornado exists as a vortex within a 3D vector field, with a sink or source at its base. The point at which the vector field vanishes, a *critical point*, can be used to find a location in each timestep that could possibly show a tornado. This narrows the search space, both in time and space.

With the exception of a few algorithms [Globus et al. 1991; Parnell et al. 1996], the algorithms used to calculate critical points share a similar trait that make them perfect candidates for modern parallel architectures: the operations applied to each sub-volume within a simulation can be performed independently of all other operations on other sub-volumes. Modern graphics hardware has been shown to accelerate calculations for parallel problems, and conventional parallel machines have been used to accelerate critical point calculations [Gerndt et al. 2006]. No work has been done to accelerate the calculation of critical points using graphics hardware.

The rest of the paper is presented as follows. The background section summarizes the information related to vector fields and critical points. The related work section is split into the two subsections used in this work: an overview of Greene's bisection method [Greene 1992], and methods of accelerating problems using parallel machines, including graphics hardware. The implementation section details the changes required to Greene's bisection to execute on modern graphics hardware. Finally, the results of benchmarks are presented, comparing the CPU version of the algorithm to a CPU+GPU version.

2 Background

This section provides background information on vector fields, critical points, and algorithms used to locate the critical points. The vector fields are created either from simulations, or from derived calculations of scalar fields.

2.1 Vector Fields

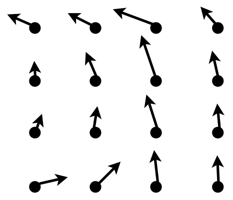


Figure 1: *An example of a 2D vector field*

A vector field consists of a grid of positions, with each position having one or more vectors. An example 2D vector field is shown in Figure 1. Vector fields are generally used in physics-based simulations from complex parallel codes. A scalar field is similar to a vector field, but with scalars associated with the positions instead of vectors. By calculating the gradient or some other function that yields a vector, a derived vector field can be created from scalar fields. Since the vector fields themselves share common properties across disciplines, algorithms applied to vector fields will benefit all domains that use vector fields. So, for example, visualization techniques created for and applied to computational fluid dynamics simulations can be applied and will benefit space weather simulations and vice versa. A more thorough description of vector fields is given by Morse and Feshbach [1953], with applications to physics problems.

2.2 Critical Points

Critical points are positions within a vector field where the vector field vanishes. More detailed information about critical points can be found in the books by Arnold [1992; 1993], and Palais and Terng [1988]. Finding these positions within a vector field is important because it allows the exploration of a vector field by extracting features from the vector field that exist near or are characterized by the critical points. These features are enough to characterize the global topology of the vector field, allowing less information to be displayed. When dealing with simulation data from complex parallel codes, minimizing the search space helps the exploration process.

The *topological degree*, also called the *Poincaré index*, is a mathematical property from the Poincaré-Hopf theorem. It is used as a value to determine whether a critical point exists within some volume of space. It represents the number of times the angle (2D) or solid angle (3D) of the vectors at the boundary of a closed line or volume, covers a unit circle (2D) or unit sphere (3D). The volume used is dependent on the algorithm, but normally a cube is

[*]e-mail: david.berrios@umbc.edu

used, since it is topologically the same as a sphere, and is non-overlapping. The two terms, degree and index, are used interchangeably. A critical point having an index of 1 represents a source, while a critical point having an index of -1 represents a sink. Higher order critical points are those critical points having an index whose magnitude is greater than 1.

3 Related Work

Several algorithms can be used to calculate critical points in vector fields; Greene's bisection method was chosen because of its ease of implementation, and quality of the results. Greene's bisection algorithm, along with other methods of calculating critical points are discussed for completeness.

In 1992, Greene published the first method for locating first-order critical points (i.e., index is -1 or 1) in 3D vector fields using the topological degree. A previous algorithm used a similar initial method [Globus et al. 1991], but found the nulls by using Newton-Raphson iteration to converge to the solution for each cell; it required an initial guess and did not always converge to a solution. Greene introduced the concept of the *topological degree* of a volume in a 3D vector field. The topological degree is the Poincare index, evaluated using surface vector values of a sub-volume within a 3D vector field. Greene's algorithm evaluates sub-cubes within a 3D vector field. For each face of a cube, a diagonal line is created that separates the face into two triangles. The vectors at the vertices of each triangle are projected onto a unit sphere. The solid angle formed by this projection is added or subtracted to a global value for a given sub-volume, depending on the sign of the cross product of two of the vectors, dotted with the third (essentially, either inward or outward). In particular, Greene focused on locating critical points with a topological degree of 1 or -1, which are critical points with a degree of one. This was also a limitation of the algorithm, since it could not find higher-order critical points (those points with degrees higher than one).

Schaeuermann et al. [1997] created an algorithm, using the notations from clifford algebra, to find singularities having a higher-order, but it only applied to 2D vector fields. Their work also included diagrams showing higher-order critical points in the 2D case.

Mann and Rockwood [2002] extended the work by Schauermann et al., and introduced an algorithm to calculate *singularities* of higher-order than previously possible for higher dimensions. Singularities, in this case, are equivalent to critical points. They utilize multivectors and operators from geometric algebra, using an implementation provided by the work of Fonijne [2006], to calculate and evaluate the topological degree of a sub-volume. For each face of a sub-cube, a grid is created and vectors are interpolated onto the grid, using trilinear interpolation of the vectors at the corners. The contributions of the trivectors and bivectors from each face are added to calculate the topological degree. As in Greene's bisection method, the cubes that have a non-zero index are bisected and the algorithm repeats. Because Mann and Rockwood's method projects the vectors onto each face of the cube, the algorithm locates singularities that exist only because of the projection of the vectors onto the faces, but some of those singularities do not exist in the original vector field. They use a heuristic to help eliminate these false singularities. Useful references for geometric algebra and its applications to computer science are included in books by Dorst et al. [2007], and Vince [2008].

Furuheim wrote a masters thesis that compares an analytical method to Greene's bisection method and demonstrates new methods of visualizing 3D vector fields using the calculated critical points [2008]. Furuheim's algorithm locates more first order critical points than Greene's bisection method, but like Greene's algorithm, it cannot find critical points of higher degree than one. This work demonstrates that Greene's algorithm is still comparable to newer techniques.

All current algorithms used to calculate critical points in vector fields rely on the Poincare index and all bisect the volume to narrow the search for the true location. The difference exists in how they setup the calculations to approximate the index.

4 Implementation

Algorithm 1 Greene's Bisection

```
 1: function calculate-degree ($Cell\ c$)
 2: $degree \leftarrow 0$
 3: for all $t \in triangles$ in $c$ do
 4:     $degree \leftarrow degree$ + compute-solid-angle ($t$)
 5: end for
 6: $degree \leftarrow degree * 1.0/(4.0 * \pi)$
 7: return $degree$
 8: end function

 9: function calculate-solid-angle($Triangle\ t$)
10: $angle_{solid} \leftarrow 0$
11: $v \leftarrow$ vectors of t
12: $\theta_1 \leftarrow \arccos((v_2 \bullet v_3)/(\text{length}(v_2)*\text{length}(v_3)))$
13: $\theta_2 \leftarrow \arccos((v_1 \bullet v_3)/(\text{length}(v_1)*\text{length}(v_3)))$
14: $\theta_3 \leftarrow \arccos((v_1 \bullet v_2)/(\text{length}(v_1)*\text{length}(v_2)))$
15: $result \leftarrow \tan((\theta_1+\theta_2+\theta_3)/4.0)*\tan((\theta_1+\theta_2-\theta_3)/4.0)*$
    $\tan((\theta_2+\theta_3-\theta_1)/4.0)*\tan((\theta_3+\theta_1-\theta_2)/4.0)$
16: $result \leftarrow \sqrt{result}$
17: if $v_1 \bullet v_2 \times v_3 < 0$ then
18:     $angle_{solid} = -a * 4.0$
19: else
20:     $angle_{solid} = a * 4.0$
21: end if
22: return $angle_{solid}$
23: end function

24: function locate-critical-points ($Cells$)
25: $criticalpoints \leftarrow \{\}$
26: for all cells $c_i$ of $Cells$ do
27:     $degree \leftarrow$ calculate-degree($c_i$)
28:     if $degree \neq 0$ then
29:         if size($c_i$) $<$ threshold then
30:             append center of $c_i$ to $criticalpoints$
31:         else
32:             $children \leftarrow$ bisect($c_i$)
33:             $childrenresults \leftarrow$ locate-critical-points($children$)
34:             append $childrenresults$ to $criticalpoints$
35:         end if
36:     end if
37: end for
38: return $criticalpoints$
39: end function
```

Greene's bisection has been implemented using C++ and NVIDIA's CUDA SDK. CUDA allows transferring data to the graphics card and performing calculations on the hardware; although it can perform almost any calculation, the problem to be calculated on the graphics hardware has to be reformulated to take advantage of the parallel architecture of the hardware. For all the cells within the simulation, cubes are created, containing the vertices at the corners and the positions of those corners. Each face of the cube is split into two triangles. The CPU version operates on each cube independently of all other cubes, while the GPU version operates on all triangles independently of all other triangles.

NVIDIA hardware can operate on 32 thread chunks, called warps, at a time. The number of threads should be multiples of 16 or 32 to maximize the number of calculations performed. Blocks are groups of threads that share memory. When threads in one block require access to data in memory (requiring a significant number of clock cycles), the graphics hardware can swap that block for another one and continue operating. As a result, the more blocks that are available to operate on a problem, the more the built-in scheduler can hide the memory latencies. For the two graphics cards benchmarked, the 8600M GT and the 8800 GT, the warp size is 32. To maximize the parallelism, each warp should have 32 threads to execute. See the CUDA programming guide[NVIDIA 2008] for more technical aspects for programming on NVIDIA hardware .

Each cube created from the data was separated into twelve triangles (two triangles per face, six faces). The grid was specified as 512 blocks $\times$ 32 threads per block for a total grid size of 16384. Each thread operated on an individual triangle, performing the series of operations listed in function *calculate-solid-angle*. Whenever possible, faster, less accurate single precision versions of operations were used. These include __tanf, and __fdividef. For each memory transfer to the graphics device, 16384 triangles are sent, and the results are then inspected. The candidate cells are created from the cells having a non-zero index; these cells are bisected into eight children, and added to a candidate cells list. When the initial pass is complete, the candidate cells are then processed and the algorithm continues. The results are returned when the cells being inspected are smaller than a threshold size.

Two versions of the GPU accelerated Greene's bisection were created: one that performed all the triangle calculations on graphics hardware, and one that only performed the initial scan on graphics hardware. In most cases, the number of candidate cells after the first pass is significantly less than the number of total cells. Very rarely will the number of critical points equal the number of cells in the 3D vector field. By eliminating the necessary overhead of transferring a smaller number of cells after each pass, the performance is increased.

5 Results

Average Performance (seconds)				
	CPU	GPU	GPU+CPU	Speedup
MacBook Pro	29.742	8.893	8.654	5.191
Mac Pro	24.307	4.834	4.682	3.437

Table 1: *Average runtimes of the three methods of calculating Greene's bisection on two different systems. Each method was executed ten times on each system. The MacBook Pro has a 2.6 GHz Core 2 Duo processor and an NVIDIA 8600M GT graphics processor. The Mac Pro has two 3.2GHz Quad-Core Intel Xeon processors and an NVIDIA 8800GT graphics processor. Only one CPU core was used for benchmarking purposes.*

To compare the performance of GPU versions to the CPU version, some assumptions need to be made. All the times are compared to the relative performance of a single core of a multicore processor. This was done because, for any given problem, more processor cores can be added to solve the problem faster. More graphics cards can also be added to solve the problem faster. To get a better sense of how a graphics processor performs relative to the base unit of a processor, comparing the graphics card performance to that of a single core makes more sense.

For all three versions, only the time spent computing the degrees of each cell was measured (function *locate-critical-points* listed above). The setup time to construct the structures necessary to pass the data between the GPU and CPU, or the time spent to open and load the data was not timed. The file tested, a space weather simulation file, contained 3077514 cells. Recent simulations contain approximately 30 million cells. The algorithm stops when the diagonal of a cell $< .005$.

The GPU version performed all solid angle calculations on the graphics card. As the stages of the algorithm progress (the depth of the breath first search), the number of completely filled blocks decreases, and the time for the memory transfers to and from the graphics card increases with respect to the calculation times. This can be seen in the performance results shown in Table 1. By only performing the initial scan of the cells on the graphics card, and completing the resulting scans with the CPU, the GPU+CPU algorithm consistently performs better than completing all the scans by the graphics card.

The accuracy of the GPU and GPU+CPU versions were virtually identical to the CPU only version, even with the use the less accurate single precision intrinsic functions __tan and __fdividef. The same number of critical points were found, with the same index values. At most, the error was $\pm \sim .01$.

6 Future Work

CUDA, as it exists on OSX, does not support 64-bit code. This remains a major limitation; the 4GB memory limit was encountered during testing of the techniques in this paper. Porting it to Linux or obtaining a 64-bit compatible version for OSX needs to be explored.

The Mann and Rockwood method has not yet been applied to real data. It is not clear if this is because it is not feasible, computationally, for real data, or because the technique would not work with perturbed or noisy vector fields. This needs to be explored. Also, a direct comparison between different methods of calculating the critical points can be presented. The parameters of all algorithms can be explored to determine which parameters affect the resulting number and quality of critical points found.

7 Conclusion

This paper presented several methods of calculating critical points in 3D vector fields. Greene's bisection method was selected, and implemented using C++ and NVIDIA's CUDA SDK. The performance on both the mobile graphics chip, as well as the desktop graphics card, show promising results. The performance on the MacBook Pro demonstrates that the graphics chip can be used to outperform both cores of the host machine when calculating a suitable parallel problem. The NVIDIA Tesla S1070 should show similar performance gains; each card alone should equal the performance of all 8 cores of the host machine. Since the system has four cards, this will be an economical and powerful way of performing complex calculations using commodity hardware.

References

ARNOLD, V. I. 1992. *Ordinary Differential Equations*. Springer-Verlag.

ARNOLD, V. I. 1993. *The Theory of Singularities and Its Applications*. Press Syndicate of the University of Cambridge.

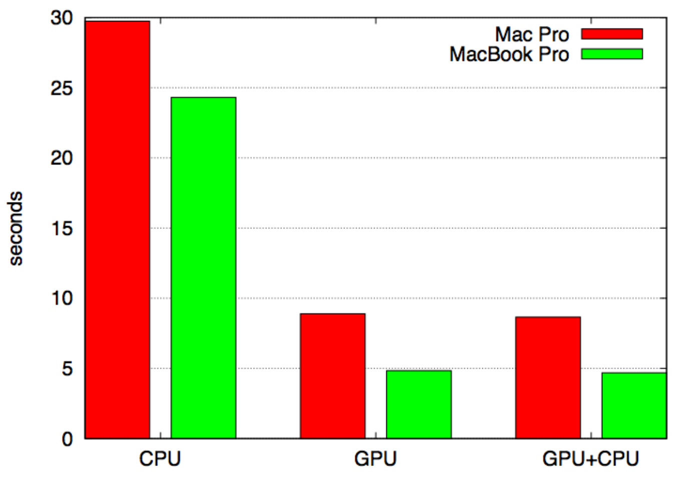
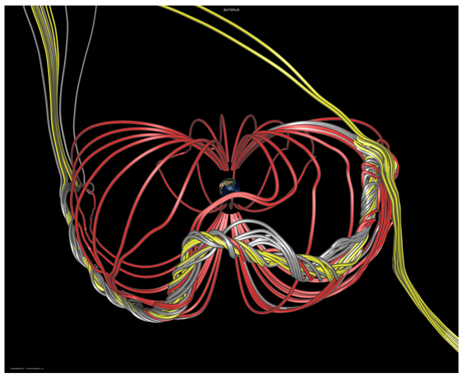
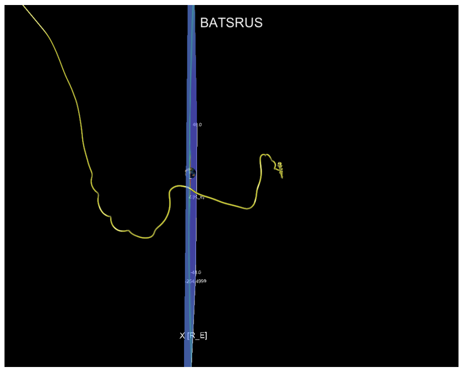


Figure 2: *Performance of the different methods of calculating the critical points using Greene's bisection. The CPU times represent the time it takes for one core to complete the calculations. The MacBook Pro has an NVIDIA 8600M GT graphics card, and the Mac Pro has an NVIDIA 8800GT graphics card. The GPU times represent the time it takes for the respective graphics cards to complete the calculations.*



(a) A fluxrope on the day-side of a global magnetosphere simulation.



(b) A single fieldline that characterizes the boundary between three different topologies.

Figure 3: *Visualizations showing a feature that can be found using critical points.*

DORST, L., FONTIJNE, D., AND MANN, S. 2007. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Morgan Kaufmann Publishers.

FONTIJNE, D. 2006. Gaigen 2:: a geometric algebra implementation generator. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, ACM, New York, NY, USA, 141–150.

FURUHEIM, K. 2008. *Classification and Visualization of Critical Points in 3D Vector Fields*. Master's thesis, University of Oslo.

GERNDT, A., SARHOLZ, S., WOLTER, M., MEY, D. A., BISCHOF, C., AND KUHLEN, T. 2006. Nested OpenMP for efficient computation of 3D critical points in multi-block CFD datasets. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, 93.

GLOBUS, A., LEVIT, C., AND LASINSKI, T. 1991. A tool for visualizing the topology of three-dimensional vector fields. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, IEEE Computer Society Press, Los Alamitos, CA, USA, 33–40.

GREENE, J. M. 1992. Locating three-dimensional roots by a bisection method. *Journal of Computational Physics 98*, 2, 194–198.

MANN, S., AND ROCKWOOD, A. 2002. Computing singularities of 3D vector fields with geometric algebra. In *VIS '02: Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA, 283–290.

MORSE, P. M., AND FESHBACH, H. 1953. *Methods of Theoretical Physics: Part 1*. McGraw-Hill Book Company, Inc.

NVIDIA. 2008. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA.

PALAIS, R. S., AND TERNG, C. 1988. *Critical Point Theory and Submanifold Geometry*. Springer-Verlag.

PARNELL, C. E., SMITH, J. M., NEUKIRCH, T., AND PRIEST, E. R. 1996. The structure of three-dimensional magnetic neutral points. *Physics of Plasmas 3* (Mar.), 759–770.

SCHEUERMANN, G., HAGEN, H., KRÜGER, H., MENZEL, M., AND ROCKWOOD, A. 1997. Visualization of higher order singularities in vector fields. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, IEEE Computer Society Press, Los Alamitos, CA, USA, 67–74.

VINCE, J. 2008. *Geometric Algebra for Computer Graphics*. Springer.