Jesse St. Charles was born in Chattanooga, TN and grew up on nearby Signal Mountain. He attended the University of Tennessee at Chattanooga, receiving a Bachelor of Science with dual concentrations in Computer Science: Scientific Applications and Software Systems and a minor in mathematics in December 2007. He is currently in his second SULI appointment with the ASER group at Oak Ridge National Laboratory. He hopes to begin a doctoral program in computer science in the fall of 2008. His research interests include emergent behavior, complex systems, self-organizing systems, and swarm intelligence.

Dr. Xiaohui Cui is an associate research scientist in the Computational Sciences & Engineering Division of Oak Ridge National Laboratory. He received his Ph.D. degree in Computer Science and Engineering from University of Louisville in 2004. His research interests include swarm intelligence, agent based modeling and simulation, emergent behavior in complex system, high performance computing, information retrieval and knowledge discovering. His current research focuses in developing new computational algorithms inspired from biological models. His research works include collective intelligence of multi-agent system, parallel and distributed knowledge discovering, swarm based social simulation, and adaptive agent cognitive modeling.

Flocking-Based Document Clustering on the Graphics Processing Unit

Jesse St. Charles, Robert M. Patton, Thomas E. Potok, and Xiaohui Cui

ABSTRACT

Analyzing and grouping documents by content is a complex problem. One explored method of solving this problem borrows from nature, imitating the flocking behavior of birds. Each bird represents a single document and flies toward other documents that are similar to it. One limitation of this method of document clustering is its complexity $O(n^2)$. As the number of documents grows, it becomes increasingly difficult to receive results in a reasonable amount of time. However, flocking behavior, along with most naturally inspired algorithms such as ant colony optimization and particle swarm optimization, are highly parallel and have experienced improved performance on expensive cluster computers. In the last few years, the graphics processing unit (GPU) has received attention for its ability to solve highly-parallel and semi-parallel problems much faster than the traditional sequential processor. Some applications see a huge increase in performance on this new platform. The cost of these high-performance devices is also marginal when compared with the price of cluster machines. In this paper, we have conducted research to exploit this architecture and apply its strengths to the document flocking problem. Our results highlight the potential benefit the GPU brings to all naturally inspired algorithms. Using the CUDA platform from NVIDIA®, we developed a document flocking implementation to be run on the NVIDIA® GEFORCE 8800. Additionally, we developed a similar but sequential implementation of the same algorithm to be run on a desktop CPU. We tested the performance of each on groups of news articles ranging in size from 200 to 3,000 documents. The results of these tests were very significant. Performance gains ranged from three to nearly five times improvement of the GPU over the CPU implementation. This dramatic improvement in runtime makes the GPU a potentially revolutionary platform for document clustering algorithms.

Introduction

Analysts are continually faced with the extremely difficult task of extracting relevant data from thousands to millions of documents at a time. This problem is exacerbated by the large quantities of data generated through the use of computing systems, information systems, and sensor systems. The need for fast, efficient document analysis has driven the research community to develop and improve document clustering methods. One method, document flocking [4], is a nature-inspired computational model for simulating the dynamics of a flock of entities. This method takes an agent-based approach and relies on emergent organization to effectively cluster documents. The effectiveness of this approach relies on the organization that arises through a group of agents interacting through simple rules. In the case of document clustering, similar documents flock together, loosely organizing themselves according to subject. This method has met with success in clustering documents quickly, performing better than traditional methods such as K-means [4]. Unfortunately it needs to be implemented on expensive cluster computers when trying to analyze more than a few hundred documents at a time. Not only are these cluster-computers expensive, but they also lack portability and are impractical in certain environments. Our research investigates the possibility of implementing this algorithm on more portable machines, thereby bringing the clustering ability to the analyst. In our work, we compared the runtime performance of sequential and parallel versions of the document flocking algorithm. Using an NVIDIA® GPU platform we saw a dramatic fivefold improvement over the sequential CPU implementation. Ultimately, we are working toward illustrating a low-cost, high-capacity parallel computational platform suitable for most naturally inspired cooperative applications.

Materials and Methods

Document Clustering

Cluster analysis is a descriptive data mining task, which involves dividing a set of objects into a number of clusters. The motivation behind clustering a set of data is to find its inherent structure and expose that structure as a set of groups [1]. The data objects within each group should exhibit a large degree of similarity while the similarity among different clusters should be minimal [2]. Document clustering is a fundamental operation used in unsupervised document organization, automatic topic extraction, and information retrieval. It provides a structure for efficiently browsing and searching text.

There are two major clustering techniques: partitioning and hierarchical [2]. Many document clustering algorithms can be classified into these two groups. In recent years, it has been recognized that the partitioning techniques are well suited for clustering large document datasets due to their relatively low computational requirements [10]. The best-known partitioning algorithm is the K-means algorithm and its variants [11]. This algorithm is simple, straightforward and based on the firm foundation of analysis of variances. One drawback of the K-means algorithm is that the clustering result is sensitive to the selection of the initial cluster centroids and may converge to local optima, instead of global ones. Another limitation of the K-means algorithm is that it requires a prior knowledge of the approximate number of clusters for a document collection. Flocking-based clustering is classified as a type of partitioning algorithm.

Flocking Behavior

Social animals in nature often exhibit a form of emergent collective behavior known as '*flocking*.' The flocking model is a biologically inspired computational model for simulating the animation of a flock of entities. It represents group movement as seen in flocks of birds and schools of fish. In this model each individual makes movement decisions without any communication with others. Instead, it acts according to a small number of simple rules, dependent only upon neighboring members in the flock and environmental obstacles. These simple local rules generate a complex global behavior of the entire flock. The basic flocking model was first proposed by Craig Reynolds [5], in which he referred to each individual as a "boid". This model consists of three simple steering rules that each boid needs to execute at each instance over time: separation (steering to avoid collision with neighbors); alignment (steering toward the average heading and matching the velocity of neighbors); cohesion (steering toward the average position of neighbors). These rules describe how a boid reacts to other boids' movement in its local neighborhood. The degree of locality is determined by the range of the boid's sensor. The boid does not react to the flock mates outside its sensor range. These rules of Reynolds' boid flocking behavior are sufficient to reproduce natural group behaviors on the computer.

It has been shown, however, that these rules alone are not sufficient to simulate flocking behavior in nature [4]. A Multiple Species Flocking (MSF) model was developed to more accurately simulate flocking behavior among a heterogeneous population. MSF includes a feature similarity rule that allows each boid to discriminate among its neighbors and only flock with those similar to itself. The addition of this rule allows the use of flocking behavior to organize groups of heterogeneous documents into homogenous subgroups.

The Graphics Processing Unit

The GPU serves as a specialized processor that is tailored to make extremely fast graphics calculations. Demands for increasingly realistic visual representations in simulation and entertainment have driven the development of the GPU. As is evident in Fig. 1, the most recent iteration of NVIDIA®'s GPU has a theoretical performance of over 100 times more floating point operations per second than the current top-of-the-line desktop CPU (the 3.0 GHz Intel Core2 Duo). This difference arose from the evolution of the GPU on highly parallel, computationally intensive calculations rather than data caching and flow control [6].

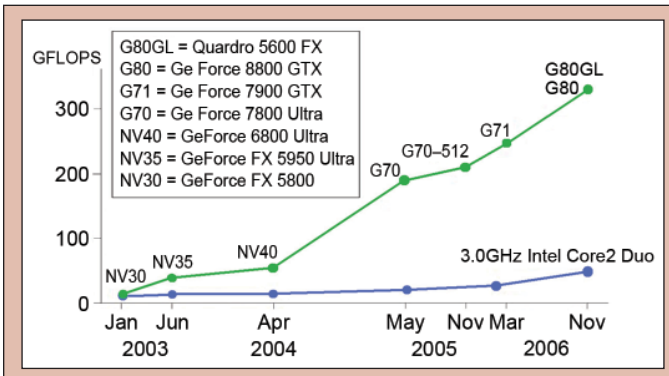

Figure 1. Floating-Point Operations per Second for the CPU and GPU [6].

The immense computational power of the GPU was noticed by developers and a move to exploit this power was made. A community of general-purpose GPU programmers quickly arose (www.gpgpu.org) and pioneered programming on the GPU. In the early stages, programming for the GPU was non-intuitive. Vertex shader languages, such as Sh, Cg, and OpenGL, were the only ones available for general use with the GPU and these focused entirely on the graphics paradigm. Consequently, they did not have appropriate naming constructs for general use and therefore were not particularly programmer friendly. Also, early GPU architectures had basic limitations that prevented some common programming operations [3]. To solve some of these problems and encourage general use of the GPU, NVIDIA® developed the GPU language CUDA as well as a more robust architecture for its GPUs.

NVIDIA® CUDA

CUDA stands for Compute Unified Device Architecture [6]. It is a C-like language that allows programmers to easily write programs to run on certain NVIDIA® GPUs. CUDA 1.0, used in this research, was released in July 2007. CUDA programs can

run using any graphics cards that use the G8x architecture [6]. Depending on the model number, members of the G8x family will have between two and sixty-four SIMD (Single Instruction stream Multiple Data stream) processors. Each SIMD processor contains eight processing elements and has access to 16KB of fast, locally shared memory, 64KB of locally cached texture memory, and 64KB of locally cached constant memory. All multiprocessors also have access to slower main device memory.

Since CUDA was developed to be run on a parallel architecture, certain parallel programming constructs and limitations are inherent to the language. Execution on this architecture is thread-based. Threads are organized into *blocks* and executed in groups of 32 threads called *warps*. Blocks are organized in groups called *grids*. All threads in a single block will execute on a single multi-processor and can exchange data through that processor's shared memory. The algorithm that is executed on the GPU directly is called a *kernel*. To run a kernel on the GPU, dimensions for the number of blocks and the number of threads per block must be specified. The unique ID of each thread and block is then used to access data unique to it. The relationship between grids, blocks, threads, and memory is illustrated in Fig. 2. A thread running on the GPU does not have access to CPU main memory. Once a kernel is run by the host (CPU), its GPU blocks all communication to and from the host, until all threads spawned by the kernel die. During kernel execution the host does not spin and can spawn additional kernels in other graphics cards present in the system.

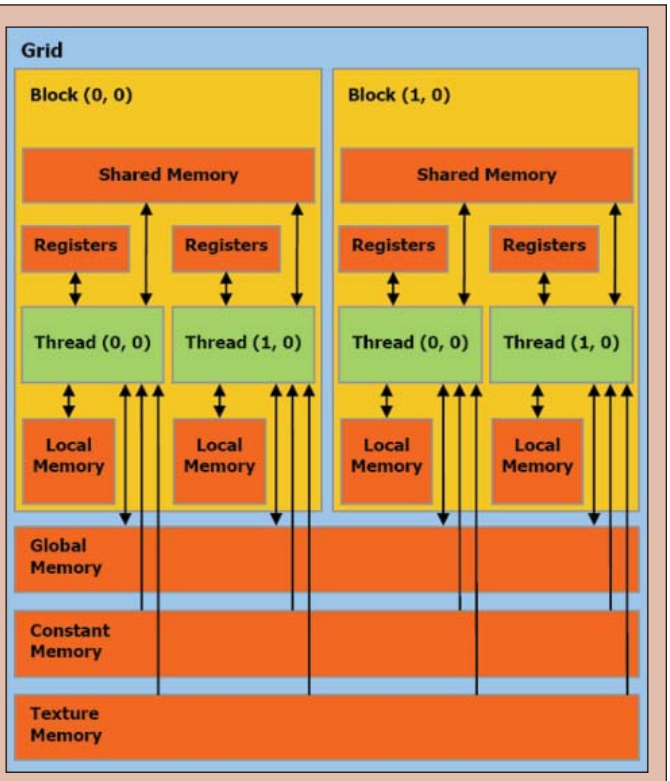

Figure 2. Thread, Block, and Grid memory relations [6].

Experimental Environment

In setting up our research we made an attempt to use low cost, commercially available equipment to help highlight the cost and performance benefits of our approach. All tests were run on a single desktop workstation, the Dell Precision 370. This machine houses 4GB of RAM and a single 3.6 GHz Intel processor with hyper-threading. We added an NVIDIA® Geforce 8800GTS graphics card to the workstation to enable the use of CUDA. The 8800GTS contains 14 SIMD processors and has 648 MB of device memory. All experiments were run under Windows XP Service Pack 2, and CUDA programs ran under CUDA 1.0.

Challenges

One fundamental challenge of programming in CUDA is adapting to the Single Program Multiple Data (SPMD) paradigm. SPMD is different from traditional parallel paradigms in that multiple instances of a single program act on a body of data. Each instance of this program uses unique offsets to manipulate pieces of that data. Data parallelism fits well in this paradigm while operational parallelism does not. Figure 6 provides a visual representation of the data flow in our implementation.

Once the programming paradigm is understood, there are additional difficulties in using the CUDA language. Since each warp is executed on a single SIMD processor, divergent threads in that warp can severely impact performance. To take advantage of all eight processing elements in the multiprocessor, a single instruction is used to process data from each thread. However, if one thread needs to execute different instructions due to a conditional divergence, all other threads must wait until the divergent thread rejoins them. Thus, divergence forces sequential thread execution, negating a large benefit provided by SIMD processing. Another limitation in CUDA is the lack of communication and, consequently, the lack of synchronization between blocks. This creates possible problems of data consistency, typical of parallel modification of singular values. Currently, all functionality must be written into the kernel code. In the future, libraries could be written for CUDA as device functions to help streamline the development process.

Debugging can be difficult in CUDA. A debug mode is available in the CUDA compiler which forces sequential execution on the CPU by emulating the GPU architecture. While this mode is useful for most general types of debugging, some errors are not exposed. The emulator cannot detect any concurrency problems as its execution is sequential. Write and read hazard behavior is undefined during thread execution on the GPU, so the programmer must be cautious to avoid these errors. While running a kernel on the GPU, no access is provided to the standard output. This effectively turns the GPU into a black box when it comes to runtime behavior.

The largest constraint for us in our work was the shortage of fast, local memory. The large amount of document information and the method of document comparison forced frequent reading from global device memory. This memory is not cached and has a penalty of hundreds of clock cycles per read associated with it.

We tried to reduce the impact of this problem by caching some document terms in shared memory for fast access. Another less costly problem we ran into was the requirement of thread divergence in the implementation. Certain conditional statements could not be avoided. This seemed to have some effect on the performance, but not a significant one when compared with the performance degradation of global memory reads.

In an effort to improve the speed of position retrieval and distance calculation, all document positions were stored in texture memory. This design decision did improve the performance of our implementation on the GPU, but it put a hard limit on the number of documents that could be compared (roughly 3,600).

Implementation

The document flocking algorithm that we used in our research was developed by Cui and Potok [4]. This approach treats documents as boids and uses the MSF model to cluster based on a similarity comparison between documents. Rather than use the feature similarity rule, we nullified the alignment and cohesion rules for documents that were not similar. Thus, for dissimilar documents, separation is the only active rule, causing them to repel one another. This algorithm was implemented in CUDA 1.0 and was run on the GPU of our test workstation. Another similar but sequential implementation was written in C and run on the CPU of the same machine.

Adapting the document flocking algorithm used in an SPMD environment is not overly difficult. We implement the algorithm in two kernels. The first kernel creates a thread for each document pair (n^2 threads in total) and compares their locations to determine if the distance between them is within the neighborhood threshold. If the distance is small enough, a document comparison is initiated. This comparison computes the linear distance between the two documents' feature vectors. If that distance is small enough, the documents are deemed similar and treat each other as flock mates. Similar documents contribute to the final velocity of each using the separation, cohesion, and alignment rules discussed earlier. Dissimilar documents contribute to the final velocity of each using only the separation rule. Once each document's influence on the rest of the population is calculated, the second kernel is run. This kernel spawns n threads, each updating the final velocity and position of a single document. Limitations are in place in this kernel to prevent velocity from changing drastically in each generation. This forces each document to make gradual turns, exposing them to a larger number of neighbors and more accurately simulating the behavior of birds. When this kernel is finished executing, a generation is finished and the cycle begins again.

Testing

We conducted testing on populations of documents ranging from 200 to 3,000 documents in increments of 200 documents. We tested each population size 30 times and then averaged the runtime of each. We used randomly generated values for the initial position and velocity of each document for each test to prevent accidental initial seeding optimization. Each test ran the flocking simulation for 200 generations. This means that documents updated their positions and velocities 200 times based on other documents present in their neighborhood. Based on our observations, 200 generations was an adequate number to allow the documents to converge into stable clusters of similar documents (Fig. 3).

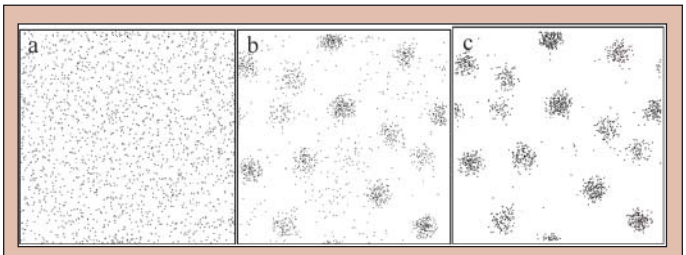

Figure 3. Snapshots of Document Flocking running on the GPU with 2,000 documents at generations 2(a), 55(b), and 200(c).

Flock Parameters

The flock parameters of each simulation were identical. The "flying" space of the documents was 300x300 units. This size space was selected to allow adequate room for each document to move. Each document had a static neighborhood radius of 30 units and a constant speed of 3 units per generation. These parameters were selected based on the flying space size and the observed behavior of the flocks. Each document had a maximum limit of a 0.35 radian deviation from its old velocity. We gave each rule a weight that encouraged system behavior typical of flocking birds. The use of these weights is described in Cui [4]. We assigned a weight of 3 to the alignment rule, 5 to the separation rule, and 3 to the cohesion rule. The document feature vector linear distance threshold was 2.50. This value was selected as it was small enough to clearly differentiate groups in the flock while not being so small that it prevented flocking altogether.

Documents

We compiled the documents used for clustering in our experiments from RSS news feeds and press releases from February 20–28, 2006 in no particular order. We initially processed the documents by stripping out HTML tags, stop words, numbers, and punctuation. We then stemmed the document content using a Porter Stemming algorithm [16]. Finally, we generated a term frequency list using TF-ICF [7] and normalized these frequencies for direct document comparison.

Timing

In the CUDA implementation, we used the timer in the *cutil* library to measure the execution time of each test. Similarly, the CPU implementation uses the Windows XP high precision timer in the *windows* library.

Results

Through our experiments we observed that document flocking on the GPU is many times faster than its CPU counterpart (see Fig. 5). We observed that with 200 documents the GPU implementation is roughly three times faster than the CPU version. As we increased the number of documents in our test set, the improvement increased. For 1,000 documents, we saw an improvement of four times over the CPU. From 1,400 to 3,000 documents the improvement levels off and remains constant at approximately 4.6 times improvement of the GPU over the CPU. Figure 4 uses a logarithmic scale to illustrate that while the performance has drastically improved the complexity of each implementation remains equivalent. The runtime of each grows at the same rate, though at different magnitudes.

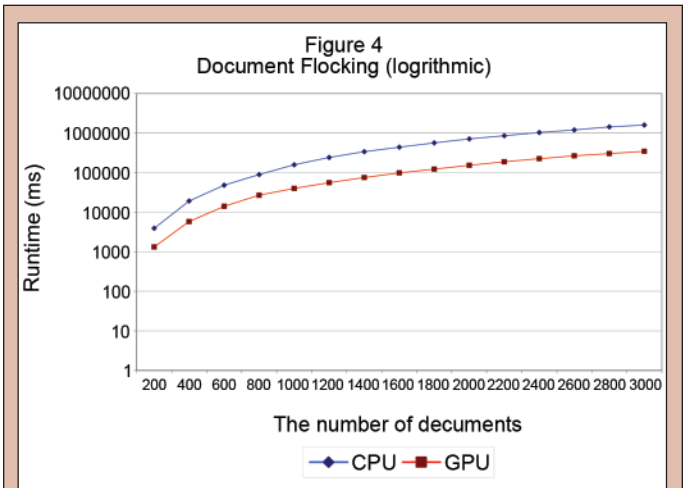

Figure 4. Document Flocking runtime, CPU vs. GPU, logarithmic.

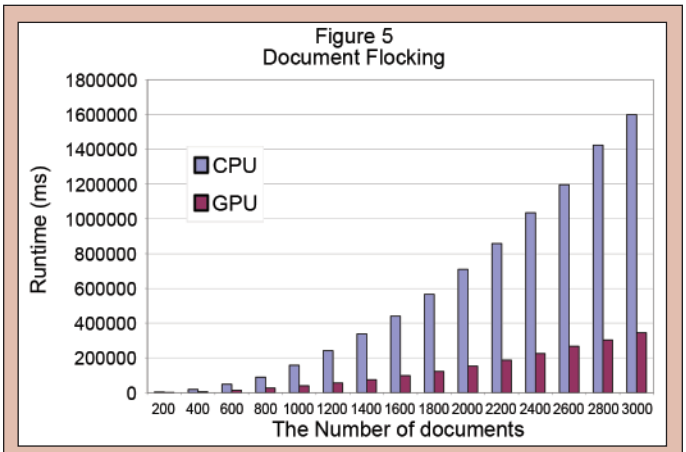

Figure 5. Document Flocking runtime, CPU vs. GPU, bar graph.

Discussion and Conclusion

The results that we have presented here add to the already substantial body of work that supports the GPU as a powerful, general computational device. This power is especially evident when applied to highly parallel algorithms. Other biologically inspired algorithms should benefit when implemented on the GPU. We believe that with continued development, document flocking on the GPU would be an extremely versatile data clustering solution. The low cost and portability of the GPU could allow analysts to cluster large data sets anywhere they are needed. The low cost could also encourage small businesses to use document clustering techniques in new ways. In future work, performance could be increased further if a faster document-to-document comparison technique was implemented. This was our most substantial bottleneck to additional performance gains. Distributing the document flocking algorithm across many GPU's could also substantially improve the number of documents that can be handled during a simulation, possibly allowing millions of documents to be clustered quickly. We did not conduct our tests on the fastest graphics card available from NVIDIA®. The currently unreleased Tesla architecture has 52 additional multiprocessors with over twice the amount of device memory. These additional capabilities would greatly enhance the already high performance we saw in our tests.

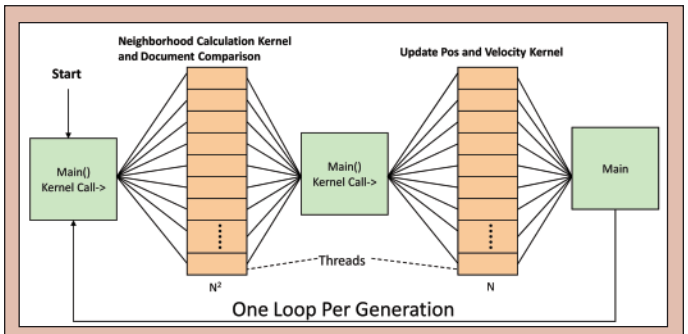

Figure 6. Document Flocking in CUDA.

Acknowledgments

Special thanks to Brian Klump, Whitney St. Charles, Ryan Kerekes, and the entire ASER staff. Also, thanks to the Department of Energy and the Office of Science for supporting me through the SULI program.

Research

Oak Ridge National laboratory is managed by UT-Battelle, LLC, for DOE under contract DE-AC05-00OR22725.

References

[1] Anderberg, M.R., Cluster Analysis for Applications, Academic Press, Inc., New York, 1973.

[2] Jain, A.K., Murty, M.N., and Flynn, P.J., Data clustering: a review. *ACM Computing Surveys*, 31 (1999), pp. 264–323.

[3] Owens, J.D., *et al.*, "A Survey of General Purpose Computation on Graphics Hardware," 2007 *Computer Graphics Forum* Volume (26), pp. 80–113.

[4] Cui, X., and Potok, T., "A Distributed Flocking Approach for Information Stream Clustering Analysis," *snpd-sawn, Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/ Distributed Computing (SNPD'06)*, 2006, pp. 97–102.

[5] Reynolds, C.W., "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics* (ACM), Volume 21, (1987), pp. 25–34.

[6] NVIDIA®, "NVIDIA® CUDA: Compute Unified Device Architecture" *NVIDIA®*, http://developer.NIVIDA®.com/cuda, Version 1.0, 2007. Accessed July 2007

[7] Reed, J., *et al.*, "TF-ICF: A New Term Weighting Scheme for Clustering Dynamic Data Streams," in *Proc. Machine Learning and Applications*, 2006, ICMLA *'06*, pp. 258–263.

[8] Fang, R., *et al.*, "GPUQP: query co-processing using graphics processors," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 1061–1063.

[9] Xu, Z., and Bagrodia, R., "GPU-accelerated Evaluation Platform for High Fidelity Network Modeling," in *2007 Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 131–140 .

[10] Steinbach, M., Karypis, G., and Kumar, V., "A comparison of document clustering techniques," KDD Workshop on Text Mining, 2000

[11] Selim, S.Z., and Ismail, M.A., "K-Means-Type Algorithms: A Generalized Convergence Theorem and Characterization of Local Optimality," *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-6* (1984), pp. 81–87.

[12] Chitty, D., "A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware," *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1566–1573.

[13] Rick, T., and Mathar, R., "Fast Edge-Diffraction-Based Radio Wave Propagation Model for Graphics Hardware," *Proceedings of ITG INICA*, 2007.

[14] Rodríguez-Ramos, J., *et al.*, "Modal Fourier wavefront reconstruction on graphics processing units," *Proceedings of the SPIE*, Volume 6272, 2006, pp. 627215.

[15] Yamagiwa, S., *et al.*, "Data Buffering Optimization Methods toward a Uniform Programming Interface for GPU-based Applications," *Proceedings of the 4th international conference on Computing frontiers*, 2007, pp. 205–212.

[16] Porter, M.F., "An algorithm for suffix stripping," *Program*, 14 no. 3, July 1980, pp. 130–137