2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

Fast Quadratic Discriminant Analysis Using GPGPU for Sea Ice Forecasting

Shadi Alawneh, Carl Howell and Martin Richard
C-CORE
Robert A. Bartlett Building, Morrissey Road
St. John's, NL, CAN, A1B 3X5
Email: {shadi.alawneh, carl.howell, martin.richard}@c-core.ca

***Abstract*—General Purpose computing on Graphics Processor Units (GPGPU) brings massively parallel computing (hundreds of compute cores) to the desktop at a reasonable cost, but requires that algorithms be carefully designed to take advantage of this power. The present work explores the possibilities of CUDA (NVIDIA Compute Unified Device Architecture) using GPGPU for Quadratic Discriminant (QD) analysis. QD analysis is a form of multivariate statistical analysis that can be applied to forecasting seasonal sea ice freeze-up and break-up. The forecast problem is formulated as a classification problem, with two classes (e.g., "ice" and "no ice") and the objective of the analysis is to decide which of the classes best describes the ice/no ice condition at a particular geographic point on a specified date. We have conducted experiments to measure the performance of the GPU with respect to the serial CPU, parallel CPU (OpenMP), MATLAB, MATLAB (Parallel for) implementations. The experiments consist of implementing a serial CPU, parallel CPU (OpenMP), MATLAB, MATLAB (Parallel for) and GPU versions of the QD analysis algorithm and executing all versions on several data sets to compare the performance. Our results show speed up of up to 426 times, reducing the elapsed time from over 15 hours to about 2 minutes.**

***Keywords*—GPGPU; CUDA; QD.**

I. Introduction

"Commodity computer graphics chips, known generically as Graphics Processing Units or GPUs, are probably todays most powerful computational hardware for the dollar. Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for General-Purpose computing on the GPU)."[1] GPUs are particularly attractive for many geometric problems, not only because they provide tremendous computational power at a very low cost, but also because this power/cost ratio is increasing much faster than for traditional CPUs.

A reason for this is a fundamental architectural difference: CPUs are optimized for high performance on sequential code, with many transistors dedicated to extracting instruction-level parallelism with techniques such as branch prediction and out-of-order execution. On the other hand, the highly data-parallel nature of graphics computations enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count.[1] Many other computations found in modelling and simulation problems are also highly data-parallel and therefore can take advantage of this specialized processing power.

Hence, in this research we are trying to use the benefit of the high performance of the GPU to implement a fast algorithm for QD analysis, which can be used in sea ice forecasting. Our goal in this paper is to study the cost of implementing a QD analysis algorithm in CUDA and its benefits in terms of performance against an equivalent CPU and MATLAB implementation.

A. Application of QD Analysis

The quantity and scale of offshore oil and gas reservoirs on the Arctic Shelf is resulting in increased activity above the Arctic Circle. Exploration and production north of the Arctic Circle comprise a plethora of challenges beyond traditional offshore operations; one notable challenge consists of a shortened operating season resulting from the presence of sea ice. When considering the sensitivity of drilling operations to the presence of sea ice, sea ice studies are paramount to ensuring safe and viable operations.

C-CORE has developed a sea ice forecasting methodology that uses a statistical model. This forecast model can simulate the timing of sea ice break-up, predict open water conditions and demonstrate its forecasting capability for operations relevant to activities in the Arctic that are sensitive to sea ice coverage. We are using the GPGPU to implement some of the numerical models in this forecasting methodology. The statistical model was developed to forecast freeze-up and break-up dates at key locations within the Arctic. Modeling the freeze-up and break-up in a given ice season follows a multi-node based QD statistical model.

II. Methodology

A. QD Approach

QD analysis is a form of multivariate statistical analysis that can be applied to forecasting. Other multivariate approaches in the literature include Principal Component Analysis and Linear Discriminant (LD) Analysis. When some form of validation data exists, as in the case here with Synthetic Aperture Radar (SAR) derived ice charts, it is recommended that a supervised pattern recognition approach be used, such as the LD and QD. The QD process is used in this paper. The forecast problem is formulated as a classification problem, with two classes (e.g., "ice" and "no ice") and the objective of the analysis is to decide which of the classes best describes the ice/no ice condition at a particular geographic point on a specified date. The mathematics of the approach is structured

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.26

to consider a large number of potential input variables that have been measured in prior years of observations and to select a subset of those variables that develops the most accurate classification decisions. The classification decision is a mathematical distance in multidimensional space from two classes, "ice" or "no ice". An unknown (potentially future forecasting) event with the minimum distance to either the "ice" or "no ice" classes will be assigned to the best fit (minimum distance) class.

The sub-sections in this section provide an overview of how the QD approach is used in this paper. Additional details on the general QD method can be found in [2], or other sources that address the use of multivariate approaches to forecasting.

1) Description and Equations: From a high level, the QD approach has two models: an ice model and an open water model. For a particular area, features that correlate with the presence of ice and open water are used to develop the QD models. Future events can then be forecasted by using the available seven day forecast environmental data. These environmental data inputs, when used in the QD models, are used to calculate the distance measured from the ice and open water models. The set of inputs that produces the minimum distance to sea ice or open water are labeled as such.

More formally, a supervised statistical based discrimination approach is presented to forecast sea ice from open water in the Arctic. This approach uses features that are extracted from sea ice charts, modeled reanalysis environmental data, and physically modeled ice thickness. These features are then used to build (or train) quadratic discriminant (QD) models representing open water and sea ice classes. Features such as air temperature, sea surface temperature, and accumulated freezing degree days are utilized in the model training phase.

The approach taken here uses grid nodes. Each node has a unique set of freeze-up and break-up models. Features that have been identified as dominant in supporting forecast optimization will be used in the training phase of algorithm development. Sea ice prior probabilities based on location and Julian day will be used as bias factors in the QD to favor the presence or absence of sea ice. The QD model is built using data from the past 20 years.

In optimizing a supervised discrimination model for differentiating open water from sea ice, three fundamental methodologies were considered. These were the discrimination function itself, feature selection (optimization), and performance evaluation. Here, an overview of these ideas is presented, including the specific methods of quadratic discriminant (QD), limited exhaustive search (LES), sequential forward selection (SFS), and re-substitution.

Bayesian decision theory is the fundamental statistical approach when solving pattern recognition problems [2]. It follows from the assumption of general multivariate normal density and Bayesian minimum error decision criteria that the maximum likelihood QD function has the form of:

$$g_i(x) = -\tfrac{1}{2}(x-\mu)^t \textstyle\sum^{-1}(x-\mu) - \tfrac{n}{2}\ln(2\pi) - \tfrac{1}{2}\ln|\textstyle\sum| + \ln(P(w_i)) \quad (1)$$

Here x is the sample column vector of length n, μ is the mean sample column vector of length n, $\sum$ is the nn covariance matrix, $P(w_i)$ is the prior probability, and $|\sum|$ and $\sum^{-1}$ are its determinant and inverse, respectively.

The QD models $(g_i(x))$ for each class are built by estimating the population mean and covariance from the known training data. This way, online classification of an unknown sample can be evaluated using the distance from each $(g_i(x))$. The class function which produces the maximum scalar value (minimum distance from class) is the class assignment for the unknown sample [3].

2) Feature Selection Methods: One of the fundamental challenges in statistical pattern recognition is to determine which features should be employed for the best classification results [4]. Feature selection can be defined as follows: given a set of candidate features, select a subset that performs the best under a classification system [5]. Feature selection algorithms will not only reduce the cost of running a classification algorithm by reducing the feature space, but can also provide a better classification model due to the statistically favored feature space that better fits the pattern recognition problem [6]. In this work, we have used the limited exhaustive search (LES).

The LES is simply the exhaustive search (ES) algorithm with a time stop criteria as opposed to the evaluation of all combinations. The LES was inspired by the fact that the lower feature spaces can be evaluated exhaustively with reasonable computation times as the bulk of computational load exist in evaluating the higher order feature spaces. As well, the LES supports limited data sets that would potentially suffer from the curse of dimensionality [3].

3) QD-LES Evaluation Method: It is important to estimate the classifier performance for evaluation and prediction purposes. Three main methods for performance estimation are re-substitution, hold out, and cross validation [4]. For re-substitution, all samples are used to train the classifier and to test its performance. The hold out method separates all samples into two groups, a training set and a test set. The cross-validation method iteratively divides all samples into two groups, a training set and a test set. For each iteration of cross-validation, a subset of data is extracted for training, and the remaining sample(s) are used for testing. The testing is such that each sample is tested only once during the entire process. The size of the testing subset can be as low as one sample. The cross-validation one sample testing method is commonly known as the leave-one-out (hold out) method [4].

The re-substitution method results in an optimistically biased estimate for performance and should only be used when the sample size is sufficiently large. The hold out method is unbiased; however, all samples are not used in the training phase and as such could decrease the overall potential for classification. The cross-validation method is essentially an unbiased measure for most applications; however, recalculation of the classification model for each sample test creates a substantial computational effort compared to the hold out and re-substitution measures [4].

Considering the cross-validation method for our work here, correlation between samples being "left out" and those included in testing needs to be given some special consideration. For example, if "leave one out" cross validation was employed,

the “leave one out” day may be highly correlated with both the before and after days. It is believed that “leave one out” in such instances would produce very similar results to that of re-substitution method. Hence, there would be substantially more work with no or limited gain.

In this work, we have employed the re-substitution method which maximizes the quality of the data when training each QD node. We have employed a quality control process that minimizes any over fitting problems that can occur with re-substitution such as selecting lower dimensionality QD’s. As well, for non-trivial class separation problems there are times that the optimized feature set actually contain a feature that supports the highest accuracy but does not support a stable algorithm. In such occurrences, these features are removed and the next highest ranking feature set is selected.

B. Stream Processing

The basic programming model of traditional GPGPU is stream processing, which is closely related to SIMD[1]. A uniform set of data that can be operated upon in parallel is called a stream. The stream is processed by a series of instructions, called a kernel [7]. Stream processing is a very simple and restricted form of parallel processing that avoids the need for explicit synchronization and communication management. It is especially designed for algorithms that require significant numerical processing over large sets of similar data (data parallelism) and where computations for one part of the data only depend on ‘nearby’ data elements. In the case of data dependencies, recursion or random memory accesses stream processing becomes not reasonable [7], [8]. Computer graphics processing is well suited to this, where vertices’s, fragments and pixels can be processed independently of each other, with clearly defined directions and address spaces for memory accesses. The stream processing programming model allows for more throughput oriented processor architectures. For example, without data dependencies caches can be reduced in size and the transistors can be used for ALUs instead. Fig. 1 shows a simple model of a modern CPU and a GPU. The CPU uses a high proportion of its transistors for controls and caches while the GPU uses them for computation (ALUs).

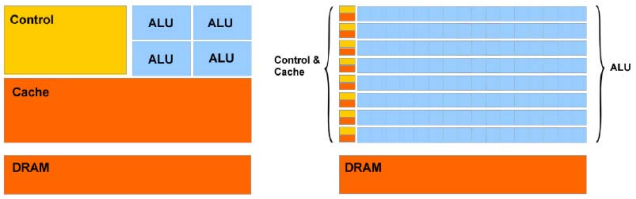

Fig. 1. Simple comparison of a CPU and a GPU [9]

C. CUDA

CUDA is a comprehensive software and hardware architecture for GPGPU that was developed and released by Nvidia in 2007. It is Nvidia’s move into GPGPU and High-Performance Computing (HPC), combining huge programmability, performance, and ease of use. A major design goal of CUDA is to support heterogeneous computations in a sense that serial parts of an application are executed on the CPU and parallel parts on the GPU[10]. A general overview of CUDA is illustrated in Fig. 2.

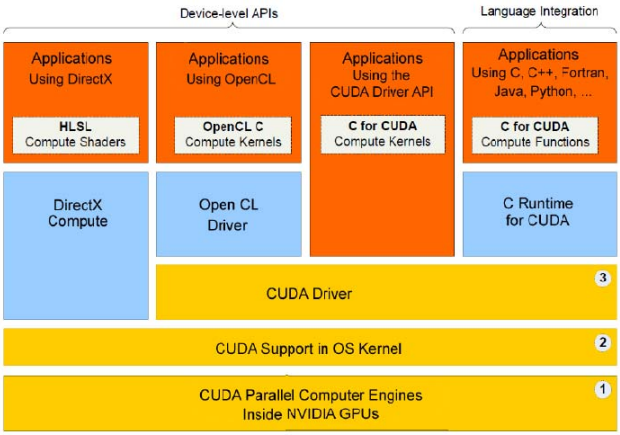

Fig. 2. CUDA overview [11]

Nowadays, there are two distinct types of programming interfaces supported by CUDA. The first type is using the device level APIs (left part of Fig. 2) in which we could use the GPGPU standard DirectX Compute by using the high level shader language (HLSL) to implement compute shaders. The second standard is OpenCL created by the Khronos Group (as is OpenGL). OpenCL kernels are written in OpenCL C. The two approaches don’t depend on the particular GPU hardware so they can be used with GPUs from different vendors. In addition to that, there is a third device-level approach through low-level CUDA programming which directly uses the driver. One advantage for this approach is it gives us a lot of control but this approach is complicated because it is low-level (it interacts with binaries or assembly code). Another programming interface is the language integration programming interface (right column of Fig. 2). As explained in [11], it is better to use the C runtime for CUDA, which is a high-level approach that requires less code and is easier to program and debug. This approach also supports other high-level languages such as Fortran, Java, Python, or .NET through bindings. Therefore, in this work we have used the C runtime for CUDA.

The CUDA programming model, as discussed in [12], suggests a helpful way to solve a problem by splitting it in two steps: Firstly into coarse independent sub-problems (grids) and then into finer sub-tasks that can be executed cooperatively (thread blocks). The programmer writes a serial C for CUDA program which invokes parallel *kernels* (functions written in C). The kernel is usually executed as a grid of *thread blocks*. In each block the threads work together through barrier synchronization and they have access to a shared memory that is only visible to the block. Each thread in a block has a different *thread ID* and each *grid* consists of independent blocks, each of which has a different *block ID*. Grids can be executed either independently or dependently. Independent

[1]Single Instruction Multiple Data, in the Flynn’s taxonomy of computer architectures

grids can be executed in parallel provided that the hardware being used supports executing concurrent grids. Dependent grids can only be executed sequentially. There is an implicit barrier that ensures that all blocks of a previous grid have finished before any block of the new grid is started. In our work, we have two kernels that train all data.

D. QDA Flowchart

Fig. 3 shows the high level flow of the QDA algorithm. At the beginning the CPU reads the point matrix data and divides it into two matrices (ice and no ice). The covariance matrix and mean for the ice and no ice matrices are calculated. Then, the number of combinations (n) is calculated based on the evaluation features space dimension (s) and number of features (nf)as shown in equation 2. The ice and no ice matrices are transferred into the GPU. The parameters for ice and no ice matrices are calculated and sent into the GPU. After that the GPU takes over the main work of the QDA. In our implementation, we have two kernels to train the ice and no ice matrices ("computeIce", "computeNoIce"). These two kernels are executed simultaneously on the GPU using streams. We have assigned one thread for each row in the ice and no ice matrices. Finally, the number of ice and no ice detected are transferred back to the CPU to calculate the accuracy of the QDA classifier. This process is repeated until the number of combinations is completed.

$$n = \sum_{i=1}^{s} \frac{nf!}{((nf-i)! * i!)} \tag{2}$$

III. Experiment Procedure

The problem explored in this paper is to develop an implementation of the QD analysis using GPGPU approach with CUDA. We have implemented a serial CPU, parallel CPU (OpenMP - 12 cores), MATLAB, MATLAB (Parallel for - 12 cores) and GPU solutions and have run all algorithms using a data set of 4 points and each point is 2880 x 51 matrix for different space dimensions (1, 2, 3, 4, 5) and we have measured the speed-up. To show the scalability of the parallel implementation we also have run the serial CPU, MATLAB, and GPU on a data set of one point with different matrix sizes (2880 x 51, 5760 x 51, 11520 x 51) for a space dimension of 4 and then we have measured the speed-up. We didn't compare the MATLAB (Parallel for) and parallel CPU (OpenMP) approaches for the data set of one point becasue they are used only to parallelize the execution of multiple points.

The performance of MATLAB is compared with CUDA in this paper to show the engineers who are using MATLAB that they can achieve a significant performance improvement from using GPGPU approach with CUDA.

We have used Intel(R) Xeon(R) CPU E5-2620 @2.10GHz and a GPU GeForce GTX TITAN Black card. This card has 2880 processor cores, 889 MHz processor clock and 336 GB/sec memory bandwidth.

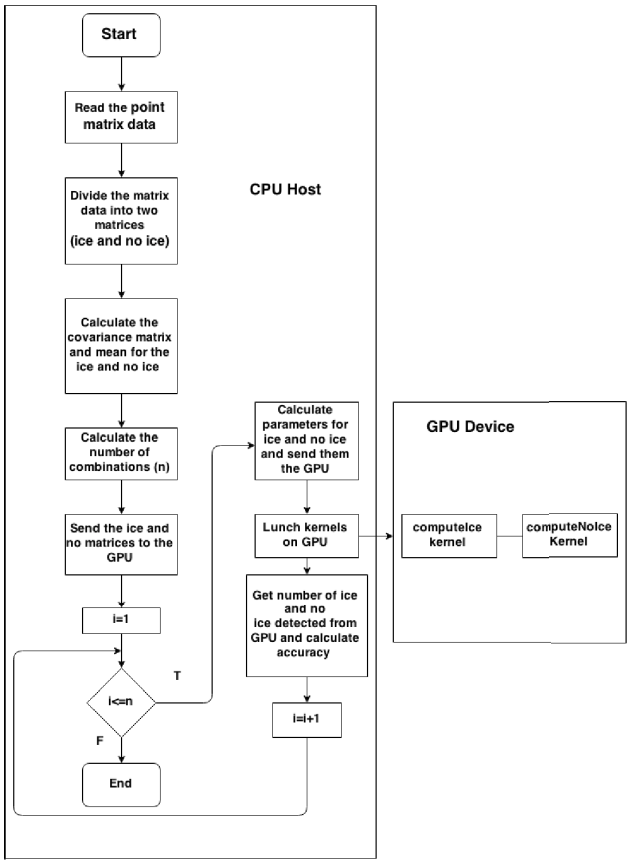

Fig. 3. QDA flowchart

A. Results

The significant performance improvement that is achieved in this paper by going from CPUs to GPUs is due to the nature of the problem (highly data-parallel) which is a good fit for the GPUs and the kernels are embarrassingly parallel.

Fig. 4 shows elapsed time of serial CPU, parallel CPU (OpenMP), MATLAB, MATLAB (Parallel for) and GPU solutions using a data set of 4 points and each point is 2880 x 51 matrix for different space dimensions (1, 2, 3, 4, 5). As we see in Fig. 4 we can tell that the GPU approach gets faster than the other approaches as the space dimension increases.

Fig. 5 shows the speed up of the GPU approach using a data set of 4 points and each point is 2880 x 51 matrix for different space dimensions (1, 2, 3, 4, 5) As we see in Fig. 5 we can tell that the speed up increases as the space dimension increases.

Fig. 6 shows elapsed time of serial CPU, MATLAB, and GPU solutions using a data set of one point with different matrix sizes (2880 x 51, 5760 x 51, 11520 x 51) for a space dimension of 4. As we see in Fig. 6 we can tell that the GPU approach gets faster than the other approaches as the matrix size increases.

Fig. 7 shows the speed up of the GPU approach using a

data set of one point with different matrix sizes (2880 x 51, 5760 x 51, 11520 x 51) for a space dimension of 4. As we see in Fig. 7 we can tell that the speed up increases as the matrix size increases.

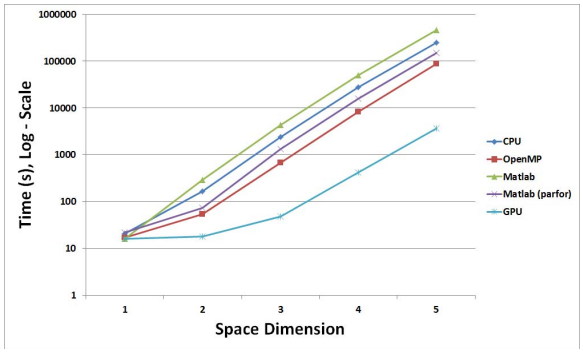

Fig. 4. Elapsed time for serial CPU, parallel CPU (OpenMP), MATLAB, MATLAB (Parallel for) and GPU solutions on a data set of 4 points.

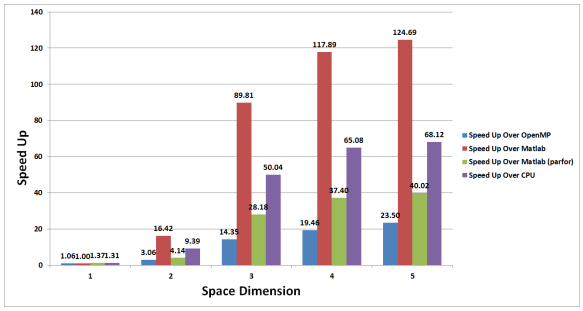

Fig. 5. GPU approach speed up using a data set of 4 points.

Fig. 6. Elapsed time for serial CPU, MATLAB and GPU solutions on a data set of one point with different matrix sizes.

IV. Related Work

Graphics Processing Units (GPUs) have a large number of high-performance cores that are able to perform high computation and data throughput. Nowadays, GPUs have support for accessible programming interfaces and industry-standard languages such as C. Hence, these chips have the ability to perform more than the specific graphics computations for which they were designed. Developers who uses GPUs to implement their applications often achieve speedups of orders of magnitude vs. optimized CPU implementations [14].

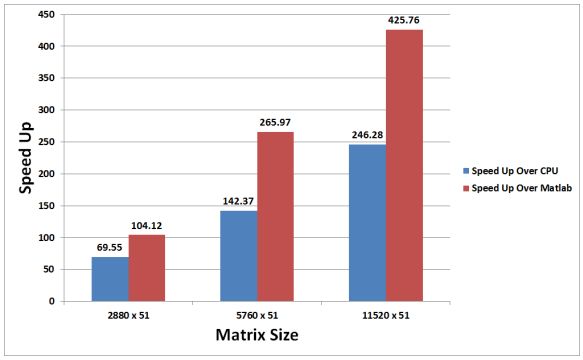

Fig. 7. GPU approach speed up using a data set of one point.

There are several advantages of GPGPU that make it particularly attractive: Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. The performance of the graphics hardware increases more rapidly than that of CPUs because of semiconductor capability, driven by advances in fabrication technology, increases at the same rate for both platforms.

QD analysis is one of the classification machine learning algorithms. GPGPU approach has been recently applied in the classification machine learning field by several researchers. Clustering strategies and the computation of a k-nearest neighbor similarity classifier is presented in [15]. A Geometrical Support Vector Machine classifier has also been implemented using GPGPU [16]. It extends different GPGPU implementations for Neural Networks [17]. In this paper, we propose a fast GPGPU implementation for the QD analysis which can be used in sea ice forecasting. We weren't able to find other state-of-the-art implementation of QD analysis using limited exhaustive search for feature selection that we can compare our GPU implementation against.

V. Conclusion

The paper introduces the basics of GPGPU and presents the stream processing programming model and the traditional GPGPU approach along with CUDA and the programming model. The experiment proved performance benefits for QD analysis. It is clear that GPGPU has the potential of significantly improving the processing time of highly data parallel algorithms.

VI. Future Work

Further development and optimization are needed for a large number of points. One way to achieve a fast QD analysis for a large number of points is to use multiple GPUs. Also, applying the QD analysis in other applications will be a next step in this research.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley, 2001.

[3] C. Howell, "Iceberg and ship detection and classification in single, dual and quad polarized synthetic aperture radar," Master Thesis, Memorial University of Newfoundland, St. John's, NL, 2008.

[4] S. J. Raudys and A. K. Jain, "Small sample size effects in statistical pattern recognition: Recommendations for practitioners," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 3, pp. 252–264, Mar. 1991. [Online]. Available: http://dx.doi.org/10.1109/34.75512

[5] A. Jain and D. Zongker, "Feature selection: Evaluation, application, and small sample performance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 2, pp. 153–158, Feb. 1997. [Online]. Available: http://dx.doi.org/10.1109/34.574797

[6] A. Jain and B. Chandrasekaran, *Dimensionality and Sample Size Considerations in Pattern Recognition Practice*. North–Holland, 1982, vol. 2, ch. 39, pp. 835–855.

[7] J. Owens, "Streaming architectures and technology trends," in *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar. 2005, ch. 29, pp. 457–470.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM Press, 2004, pp. 777–786.

[9] Nvidia, "Cuda programming guide v2.3.1," 2009.

[10] ——, "Cuda development tools v2.3. getting started," 2009.

[11] ——, "Cuda architecture overview v1.1. introduction & overview," 2009.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, March 2008.

[13] Nvidia, "Geforce gtx titan black," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/product-images.

[14] "Gpgpu website," http://www.gpgpu.org/.

[15] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, June 2008, pp. 1–6.

[16] M. Wolfe, "Implementing the pgi accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735697

[17] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, "Gpu-based simulation of spiking neural networks with real-time performance amp; high accuracy," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, July 2010, pp. 1–8.