

# Discriminant Adaptive Nearest Neighbors Classification Implementation with OpenCL and OpenMP

---

Jonathan Lisic

## Introduction:

Discriminant Adaptive Nearest Neighbors Classification or DANN, is a classification method that uses nearest neighbors classification within a localized neighborhood adjusted for between class variance and sphered through pooled variance. This Classification method is particularly useful for supervised learning problems where boundaries between two populations are nonlinear or noise variables are present. (1)

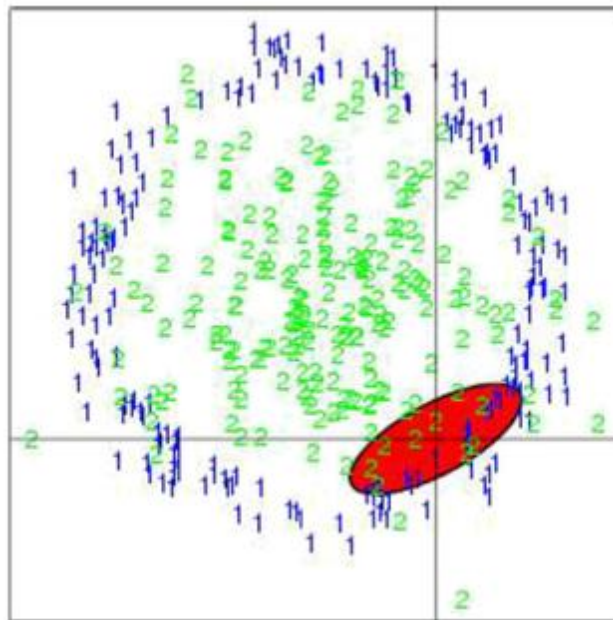


Figure 1: The red area is a DANN neighborhood located around a Query Point (2)

The basic algorithm as originally described by Hastie and Tibshirani in 1996 (2) was a five step iterative process. Empirical testing by Hastie and Tibshirani showed that the iterative step provided little gain in accuracy so it is dropped here:

1. Calculate a nearest neighborhood about the query point (Euclidian distance is generally used).
2. Calculate the Within and Between-sum-of-squares for this neighborhood ( $W$  and  $B$ )
3. Define the metric  $\Sigma = W^{-1/2}(W^{-1/2}BW^{-1/2} + \epsilon I)W^{-1/2}$ .
4. Create a new neighborhood using  $\Sigma$  and classify by nearest neighbors.

Computation of the method can be fairly laborious for a large number of query points, or a large feature space (high dimension). Each query point requires the computation two covariance matrices and the eigenvectors and values for the Within-sum-of-squares matrix. However, there is nothing inherently serial about any of these calculations. Therefore it seems natural to tap into advances in parallel computing to speed up this algorithm.

### **Motivation:**

The motivation behind optimizing this classification problem began with this semesters CSI 991 seminar in computational statistics. Professor Sutton from the Statistics Department had an implementation of this code that ran particularly slow. Because of this, I became interested in finding ways to speed up the code.

Using Rprof, a part of R's utility package, I was able to use the gprof like output to determine that the root cause was the excessive computation of two distance matrices. After fixing this issue I started looking into other ways to increase performance. One way was through matrix approximations, but this was only helped for extremely high dimensional problems.

Profiling the code for reasonable numbers of variables, it appeared that transposes and matrix multiplication took up the greatest amount of time. In addition some of my adjustments to the original R code required looping, something that R is quite bad at. So the next logical step was to move to compiled code and look into ways to minimize the cost of transposes and matrix multiplication.

Although BLAS libraries were available, I was more interested in turning this problem into an educational experience by determining if it's possible to write the code from scratch and then use parallel computing to obtain reasonable performance gain.

Two parallel methods were chosen, OpenMP and OpenCL. OpenMP was chosen simply because it's easy to use and performance should scale linearly with each additional processor. OpenCL was chosen because it runs on a large variety of CPU's and GPU's, and offers the greatest performance gain but is significantly more difficult to use.

### **Approach:**

Three Approaches, Serial, OpenMP, and OpenCL were implemented using the same algorithms. The Serial Method provided the basis for the iterative loop for OpenMP and all Computation for the OpenCL method except for the Eigenvalue computation. All Eigenvalue computations used the QR method with Householder transformations for orthogonalization. To ensure equal comparisons all calculations were done with single precision floating point arithmetic.

### **Serial Method:**

The serial method was written from scratch entirely in C. All matrix functions to calculate the DANN metric were developed and implemented through row-major matrices. Due to time constraints the program was designed for only two populations, although it would be fairly easy to extend to multiple populations.

The program takes in three data sets, the first containing all the points in the first population, the second data set similarly contains all the points in the second population, and the third data set contains all the query points to classify into one of the two populations. The program also takes user input from the command line to determine the size of the initial and secondary neighborhood.

The program then performs the iterative steps of the algorithm, constructing a new sigma for each query point and returns the classification to standard out. The results were compared against the original R code, and comparable results were attained.

Profiling the code on Linux through the use of gprof indicated that most of the processing time was spent in the `Matrix_MatrixMultiply` function and the `Matrix_MatrixMultiplyTranspose` function. Unfortunately that is all the information gprof provided. Using Shark under OSX it was possible to look at individual function calls and quickly determine that around 85% of the runtime was spent computing the eigenvalues and eigenvectors of the W matrix. The next most significant computations were other calls to `Matrix_MatrixMultiply`.

### **OpenMP:**

The next approach was to simply evaluate the query points in parallel through OpenMP. This is a trivial solution to increasing performance for any embarrassingly parallel problem on shared memory systems. Implementation was slightly problematic since the large number of variables within the main function. To remedy this situation most of the main function was wrapped in a new function and called by a loop from the original main program.

### **OpenCL:**

OpenCL programming is fairly foreign when compared to general purpose computing with multiple CPU's. To help understand this material the NVIDIA OpenCL programming guide was used as a reference, as well as a number of excellent tutorials and podcasts at MacResearch.com. However, I found AMD's documentation to be quite poor and fairly confusing, focusing primarily on hardware and little on implementation. (3) (4)

An OpenCL program requires a number of steps to run. First a context must be created; the context describes a means to communicate with an OpenCL devices such as a single or multiple CPUs, GPUs, or FPGAs; however the focus of this discussion will only be on GPUs. The context also retains and manages memory allocated on the device, and allows for reading and writing to this memory. The creation of the context is done through the OpenCL function `clCreateContext` shown below (5):

```
clCreateContext (  
    cl_context_properties * properties,  
    cl_uint num_devices,  
    const cl_device_id * devices,  
    void (*pfn_notify) (const char *errinfo, const void *  
        private_info, size_t cb, void * user_data),  
    void * user_data, cl_int*errcode_ret
```

); (6)

Allocating memory for use by OpenCL is done through the use of a special malloc function `clCreateBuffer`. Other functions exist such as `clCreateImage2D` and `clCreateImage3D`, that are useful for more specific applications. Reading memory is done through `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, both of these commands support synchronous and asynchronous operations. Reading and Writing do require the addition of a queue to perform their tasks.

```
clCreateBuffer(
    cl_context context,          // Context Writing To
    cl_mem_flags flags,        // READ/WRITE Memory to Create
    size_t size,               // Size of Buffer To create
    void *host_ptr,           // Pointer on Host to Device Memory
    cl_int * errcode_ret      // Return Code on Error
);

clEnqueueWriteBuffer(
    cl_command_queue command_queue, // Queue to process this
    cl_mem buffer,                 // Buffer on device to write
    cl_bool blocking_read,        // CL_TRUE to block
    size_t offset,                // Memory offset
    size_t cb,                   // Size of memory to write
    void * ptr,                  // Host memory to write
    cl_uint num_events_in_wait_list, // Number of events to wait
    const cl_event * event_wait_list, // Events to wait before exec
    cl_event * event             // Event assigned
);

clEnqueueReadBuffer(
    cl_command_queue command_queue, // Queue to process this
    cl_mem buffer,                 // Buffer on device to read
    cl_bool blocking_write,       // CL_TRUE to block
    size_t offset,                // Memory offset
    size_t cb,                   // Size of memory to read
    void * ptr,                  // Host memory to read to
    cl_uint num_events_in_wait_list, // Number of events to wait
    const cl_event * event_wait_list, // Events to wait before exec
    cl_event * event             // Event assigned
); (6)
```

To submit commands to the device within a context we use a queue. The queue is specified for all kernels, reading and writing to device memory, and supporting functions such as barriers for synchronization. The OpenCL function, `clCreateCommandQueue`, creates the queue:

```
clCreateCommandQueue(
    cl_context context,          // Context to attach the Queue to
```

```

    cl_device_id device,          // Device within context to use
    cl_command_queue_properties, // Additional processing modes
    cl_int * errcode_ret
); (6)

```

Once an operation is enqueued it runs immediately and asynchronously, however to enable synchronous communication OpenCL provides `cl_events` data types and the functions `clWaitForEvents` and `clEnqueueWaitForEvents`. Here `cl_events` work as tokens allowing subsets of a group of operations to work together while preserving dependencies. To create a barrier over an entire queue though would be quite cumbersome using `cl_events` so OpenCL provides `clEnqueueBarrier`. (3)

The meat of OpenCL programming comes from the kernels, or small programs that run on the GPU. Kernels in OpenCL are however not always in correspondence with programs in OpenCL, since a kernel is an instance of a specific program coupled with explicit arguments. Programs instead are a more general form of kernel and are either compiled during runtime from strings containing the program or read in as pre-compiled binaries. Programs are created specifically for the devices described in the context as seen through the function call parameters:

```

clCreateProgramWithSource(
    cl_context context,      // Context to use with this program
    cl_uint device,         // Device to compile for
    const char ** strings,  // Strings containing Kernel
    const size_t * lengths, // Length of String
    cl_int * errcode_ret    // Return error code
); (6)

```

Once a program is compiled it can then be linked against its arguments to create a kernel. This kernel can be enqueued for immediate processing by the GPU. Programs are written in OpenCL following general C guidelines allowing and providing a number of common mathematical functions and the creation of structs. Parameters including variables both from global and local memory may be passed in as arguments to the program, where global memory is the slower memory resident on a video card, while local memory is the fast on-chip memory on the video card. The following is an example of a simple OpenCL program for a Kernel.

```

__kernel void return_global_id ( __global int * GPUarray) {

    GPUarray[get_global_id(0)] = get_global_id(0);

}

```

In this problem each entry of `GPUarray` is set to the index of the problem size. A similar program using `get_local_id(0)` would most likely return considerably different results. This is due to how OpenCL performs computation. In OpenCL the entire amount of work is broken up into work groups, and each individual work group contains a set of work items. Each work item within each work group will run a

separate instance of the kernel sharing local memory with other members of the same work group and global memory with any other kernel running in the same context. Each work item also has access to private registers that store any variable declared without the `__local` or `__global` prefix. This private and local memory is usually on the order of 8k to 16k depending on work group size and hardware. Going back to example program for the Kernel, the `get_global_id(0)` function gets the index of the total work that needs to be done, while `get_local_id(0)` gets the index of the work item within a work group. Other functions exist to get the local work group and the total number of work groups being used. Since total work is divided into work groups the total work must be a multiple of the work group size.

Creating functioning OpenCL kernels was the most difficult portion of the OpenCL implementation of this problem. Since the slowest portion of the OpenCL code was the eigenvalue and eigenvector computation, only this portion of the code was performed in OpenCL. This proved to be a laborious endeavor due to hardware issues and lack of debugging tools for ATI/AMD video cards running under OSX.

Performing the QR method for eigenvalue computation is fairly simple. Given a matrix A you can compute the eigenvalue and eigenvectors for this matrix by performing a QR decomposition on A and then multiplying the decomposition as RQ. This new matrix will then be used for A during the next iteration. This creates an implicit inverse power method iteration for all eigenvector/value pairs within the original matrix A. Through successive iterations the new A's will converge to a diagonal matrix containing the eigenvalues along the diagonal; by accumulating the Q matrices through multiplication the eigenvalues can be obtained.

#### *QR Iteration Algorithm (8):*

1.  $A_0 = A$
2. FOR  $k = 1, 2, 3, \dots$
3.  $Q_k R_k = A_{k-1}$
4.  $A_k = R_k Q_k$
5. END

The only difficult portion of the QR method for eigenvalue computation is the computation of the QR decomposition. This decomposition can be done in a number of ways including Gram-Schmidt orthogonalization, Householder reflections, or Givens rotations[3]. The Householder method was implemented here, but other methods could have been easily used as well. (7) (8)

The Householder method decomposes a matrix into an orthonormal set Q, and an upper triangular matrix R. This is done by first reflecting the first column vector of A onto the first axis through a householder reflection. This is repeated for a sub matrix of A excluding the first row and column of the prior reflected original matrix. This is continued for all but the last column which is a scalar. The resulting reflected matrix A will be the upper triangular matrix R, while the accumulation through multiplication of the householder reflections will be the orthonormal set Q. (8)

#### *Householder Orthogonalization Algorithm (8):*

1.  $A_1 = A$
2. FOR  $k = 1$  TO  $n - 1$
3.  $\|A_k[:, k]\|_2$  // Calculate  $l_2$  norm of matrix  $A_k$ 's  $k^{\text{th}}$  column vector
4.  $a = -\text{sign}(A_k[k, k])$  // Get the opposite sign of the diagonal from  $A_k$
5.  $v_k = A[:, k] - a(\|A[:, k]\|_2)I[:, k]$  // Calculate the vector for the Householder transformation where  $I$  is the identity matrix
6.  $H_k = I - 2(v_k v_k') / (v_k' v_k)$  // Compute the Householder Matrix
7.  $A_{k+1} = H_k A_k$
8. END

Accomplishing this algorithm under OpenCL required a number of kernels for each portion of the program:

#### *MatrixCopy*

The Matrix Copy kernel program is a fairly simple. It takes the contents of a given Matrix and copies it to another matrix of equal size. Blocking is done after global id 0 copies the width and height. This function can be improved in a number of ways, but the simplest would be to just use the built in Memory Copy function that I only recently discovered.

#### *GetColumnVector*

GetColumnVector is a kernel program that gets a column vector from a matrix of a given index. Since the Matrices are row-major any access to a column vector will be uncoalesced. Coalesced writing was performed through the use of a temp variable and a barrier. This program also handles the issue of creating a sub matrix by setting leading indecencies of the resulting matrix to zero for prior iterations.

#### *InnerProduct*

The Inner Product kernel program takes a vector and calculates both the norm and inner product of the vector. The implementation of this program is fairly interesting and uses a technique from the NVIDIA OpenCL programming guide. Unfortunately unlike the example in the guide it is not possible to statically define local variables through kernel arguments, so local variables are defined within the kernel program.

This program operates strictly within a single workgroup, although a larger problem size may be specified. Each work item in this workgroup performs a coalesced copy to a local array. After this a reduction technique is performed where a stride is created equal to one half of the workgroup size. This stride is used to associate pairs of work items together, where one work item has an id less than one half of the total workgroup size and the sum of this id with the stride is equal to another id greater than one half the total workgroup size. In this algorithm the lower id will then sum the contents of it's array entry with that of the higher id's array entry. This is then continued by following the same steps except the stride size is reduced by one half each time. Since stride is an integer the result will eventually be a

zero signaling the termination of the reduction where the  $0^{\text{th}}$  entry will have the total sum in  $(\log_2 k) + 1$  iterations.

Unfortunately the algorithm fails for non integer multiples of two so a block size is defined that causes the reduction to occur as if the workgroup size was a power of 2. However this does require editing the text file for optimal values larger than the chosen workgroup size.

### *AdjustVector*

Adjust vector is a simple program that creates the  $v_k$  matrix described earlier for the Householder transformation. To accomplish this task it reads in the original vector and its norm, and then modifies the vector index associated with the current iteration by either subtracting or adding the norm to it as described by the Householder algorithm described earlier.

If the norm is zero the iteration is typically skipped, unfortunately branching is difficult to do with kernels so instead an identity matrix will be created through the outer product kernel, assuming that the vector passed to it is all zeros. So here we create a vector of all zeros.

This is a fairly efficient program since the vectors are entirely located in adjacent memory.

### *OuterProduct*

The OuterProduct kernel program is fairly simple in function, but fairly complex in implementation. This program calculates the outer product of a vector. It accomplishes the task by reading in a portion of the vector to local memory then iterating over each element of the array from global memory, multiplying the local portion of the vector by each proceeding element. The product is then written to the resulting Matrix.

The method is fairly efficient, but locally storing coalesced portions of the array to reduce memory transfers would be optimal. The initial implementation tried this, but for unknown reasons the second array overwrote the original matrix, so this suboptimal method was used.

### *MatrixScalarDivideSubtract*

This function calculates the Householder reflection given the inner and outer product. If the inner product is zero it will simply return the identity matrix. Since all operations are equally done to each element the kernel is efficient as possible without combining its operations with another kernel.

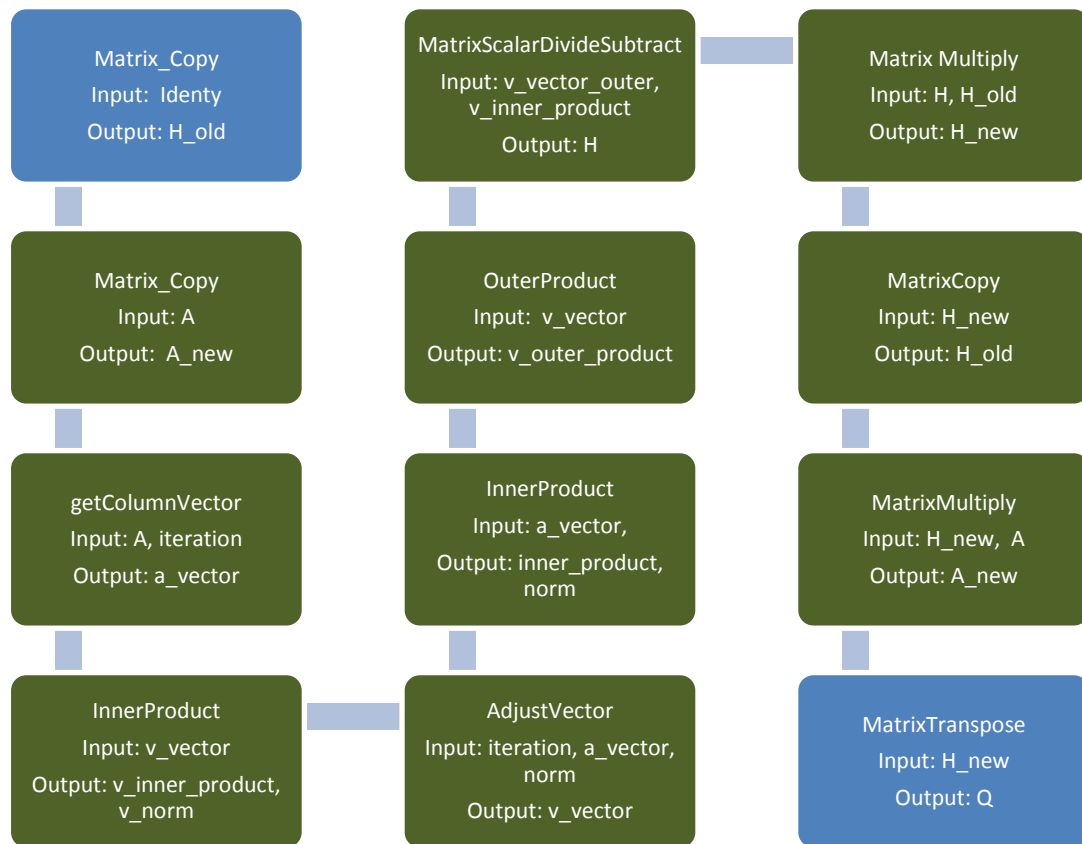
### *MatrixMultiply*

The Matrix Multiply kernel program was based off the NVIDIA example provided in their OpenCL programming Guide. This method is simple, but entirely inefficient, but mirrors the same method used on the GPU. The reason for the inefficiency is that for each pair of elements used for calculating the inner product, a new read must occur. Therefore there are two reads from both of the matrices per iteration, and since we are using row-major form for these matrices only the second matrix has coalesced reads. Fortunately when the data is written out, it is entirely coalesced.

## MatrixTranspose

This kernel program is an unoptimized matrix transpose that reads elements of one Matrix and reverses the coordinates in writing to the resulting array. There are many more optimized methods to perform this task, the simplest would be to reduce the number of reads from global memory by reading to local memory, performing the transform, and writing back out. Unfortunately due to the nature of the problem either the read or write must not be coalesced.

The kernels were assembled in the following order to compute the eigenvector, where green blocks are iterated over  $n - 1$  times:



Upon completion of this process the  $A_{\text{new}}$  matrix is used as  $R$ , combined with  $Q$  from the last step the eigenvalue and vector computation can continue.

No optimization was done to the OpenCL kernels due to lack of time. Ideally the worksize and local size should be decoupled from the size of the matrix to improve performance through padding. In addition the number and placement of barriers should be re-evaluated and local memory should be used more often, particularly in the passing of data between kernels. A good deal of speed up could be attained by simply replacing the current Matrix Multiply with one that uses local memory caching instead of direct global memory usage.

## Results:

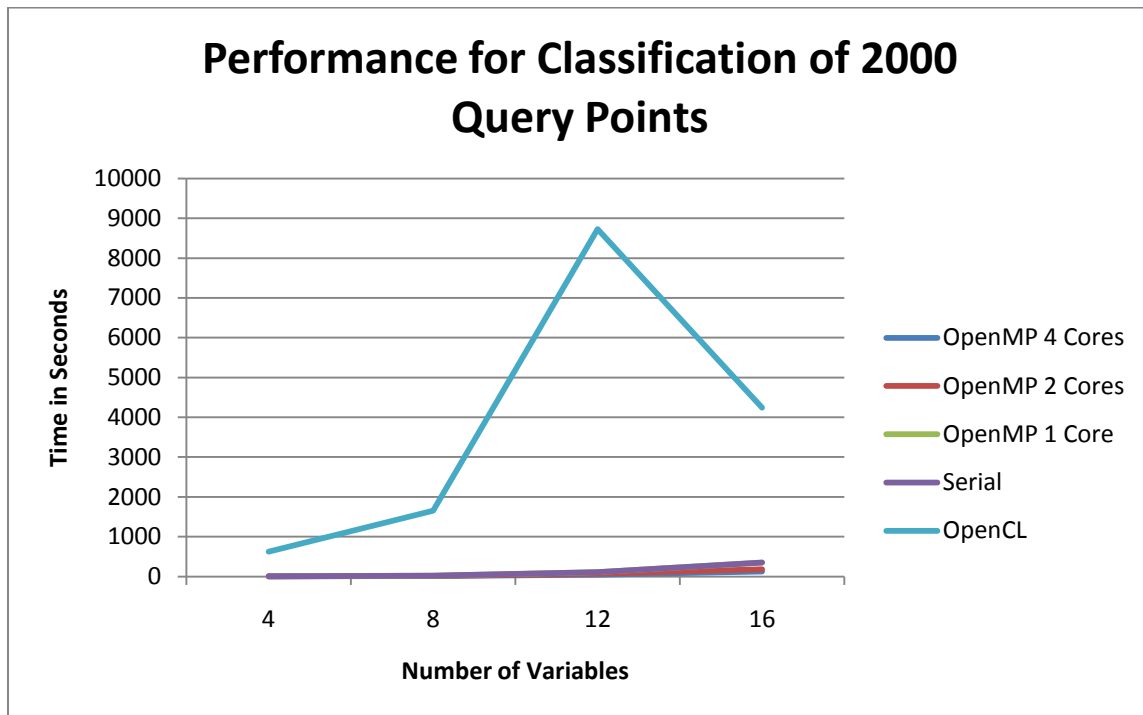
The following are timings from the three C based implementations for a variety of variable sizes using the same data and computer.

Performance for Classification of 2000 Query Points (Time in Seconds)

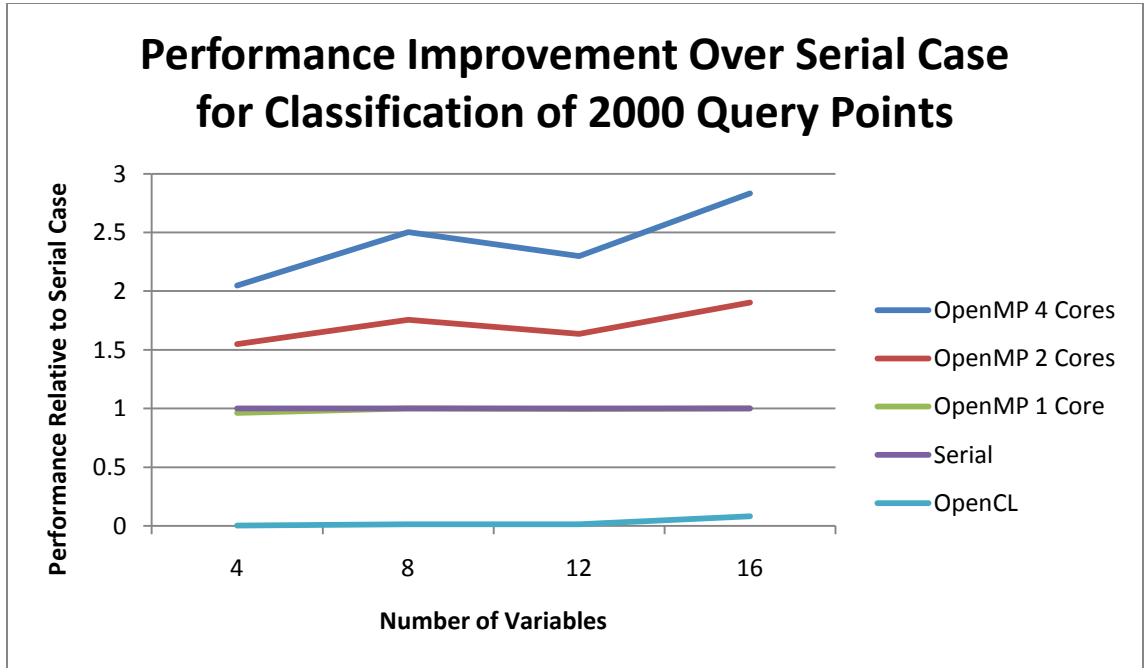
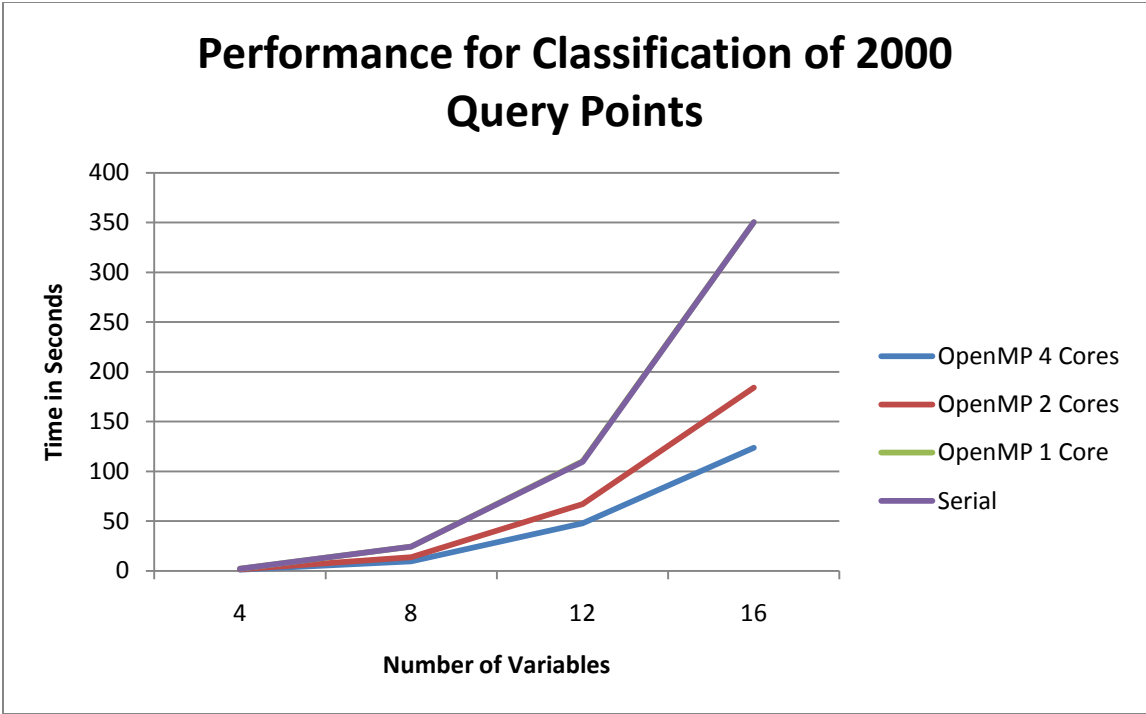
		4 Variables	8 Variables	12 Variables	16 Variables
R		38.810	40.307	N/A	N/A
OpenMP	4	1.069	9.68	47.714	123.75
	2	1.414	13.795	67.028	184.159
	1	2.275	24.206	110.118	350.368
Serial		2.189	24.239	109.644	350.52
OpenCL		626.55	1656.734	8725.735	4243.323

Epsilon = 1, 100 Training points for each population, Initial Neighborhood 10, Secondary Neighborhood 5  
R Implementations crashed with over 10 variables

To visualize this distance a chart was provided.



Due to the excessive amount of time it took to perform OpenCL computations a secondary chart is provided without the OpenCL Data.



**Analysis:**

The most glaringly obvious observation is that the OpenCL code is slow, in fact amazingly slow. This clearly shows that unoptimized GPU code will most likely not perform well. In addition, what is also obvious is that a local work size of twelve is considerably non-optimal for the ATI 4870 that the test was

run on. By AMD's documentation a wave front, similar to NVIDIA's half-warp, work best in multiples of 16.

The serial and OpenMP implementations run as expected for the most part. The eigenvalue computation performed here should be of the order of  $O(n^3)$  which seems a bit higher than what we are seeing here, but we are only working with small values of  $n$ . No compiler flags optimizations were used, for either implementation so adding these in would most likely increase performance considerably.

Another issue of note was that not all results produced by each method were identical. Although the Serial and OpenMP results were identical, the R versions were computed using double precision arithmetic and the OpenCL version computations were slightly different in implementation than the serial version creating slight differences in eigenvalues. These differences therefore can change the ranking of close values from different populations changing the nearest neighbor results. There were also issues with stability with the eigenvalue and eigenvector computation causing small eigenvalues to underflow, since the inverse of the eigenvalues were needed the small eigenvalues were approximated by the next smallest eigenvalues.

### **Conclusion:**

The serial and OpenMP based implementations provided here provide significant improvement over the original R code in performance. The serial code performed noticeably better than the R code highlighting some of R's weaknesses in computationally sensitive tasks. The OpenMP code performed very well and scaled nearly close to linearly with the number of cores available, although the gains decreased slightly towards four cores. This improvement is as expected since there is no dependence between each query point evaluation. The OpenCL implementation on the other hand performed very poorly, it was even considerably worse than the original serial and R code it was meant to optimize.

The lack of performance for the OpenCL implementation was primarily due to my inexperience writing in OpenCL, the lack of good or any debugging and profiling tools for AMD/ATI hardware under OSX, and lack of time. However, it was a very good learning experience on how to perform computations with OpenCL and with future work I hope to greatly increase the speed of this algorithm on GPU's.

## References:

1. **Hastie, Trevor, Tibshirani, Robert and Friedman, Jerome.** *The Elements of Statistical Learning Data Mining, Inference, and Prediction 2nd Ed.* Stanford : Springer , 2009.
2. *Discriminant adaptive nearest neighbor classification.* **Hastie, Trevor and Tibshirani, Robert.** 1996, IEEE Trns Pattern Anal Machine Intelligence, pp. 607-616.
3. **NVIDIA Corporation.** *OpenCL Programming Guide for the CUDA Architecture Version 2.3.* Santa Clara : NVIDIA, 2009.
4. **AMD.** *ATI Stream Computing OpenCL Programming Guide.* Sunnyvale : Advanced Micro Devices Inc., 2010.
5. **Khronos Group.** *The OpenCL Specification, Version 1.0, Document Revision 29.* Beaverton : Khronos OpenCL Working Group, 2008.
6. —. OpenCL API 1.0 Quick Reference Card. s.l. : Khronos Group, 2009.
7. **Heath, Michael T.** *Scientific Computing An Introductory Survey 2nd Ed.* New York : McGraw-Hill, 2002.
8. **Wikipedia.** Wikipedia: QR decomposition. *Wikipedia.* [Online] May 05, 2010. [Cited: May 05, 2010.] [http://en.wikipedia.org/wiki/QR\\_decomposition](http://en.wikipedia.org/wiki/QR_decomposition).
9. —. Wikipedia: OpenCL. *Wikipedia.* [Online] May 05, 2010. [Cited: May 05, 2010.] <http://en.wikipedia.org/wiki/OpenCL>.
10. **Gohara, David W.** MacResearch OpenCL Tutorials. *macresearch.org.* [Online] August 24, 2009. [Cited: May 05, 2010.] <http://www.macresearch.org/opencl>.
11. *Bayesian Adaptive Nearest Neighbor.* **Guo, Ruixin and Sournak, Chakraborty.** 2010, Statistical Analysis and Data Mining, pp. 92-105.