High Performance Programming with Multicore & GPU
Final Project
Professor Herbordt

Luke Sorenson   Ryan Lader   Katie Lewis

May 5, 2016

1 Introduction

Throughout this course, we have studied several algorithms and methods of obtaining optimal performance by using optimization techniques - serial optimizations, multithreading, and GPU. For this project we optimized two learning algorithms: Perceptron, and K-Nearest Neighbors. Perceptron works as a supervised machine learning algorithm for binary classifiers. The algorithm determines whether an input belongs to a distinct class A or B. Perceptron works to perfectly separate data or until convergence occurs. KNN, on the other hand, is an instance-based or "lazy learning" algorithm that compares every new piece of data to all of the labeled data, computing some distance between the points, and determining its class based on the nearest labeled data. For Perceptron, we were wrote a serial implementation in C based off MATLAB code, multithread it using PThreads, and transfered the code onto the GPU using both CUDA and OpenCL. K-Nearest Neighbors was written serially in C++ and multithreaded using PThreads as well. These algorithms lend themselves to parallelization because they are highly iterative and perform independent operations on the training data.

For all serial and multithreaded tests performed for this report, we used the following machine specifications:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    1
```

```
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Stepping:              7
CPU MHz:               1200.000
BogoMIPS:              4999.29
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              15360K
NUMA node0 CPU(s):     0-5
NUMA node1 CPU(s):     6-11
```

2 Perceptron - An Overview

The Perceptron algorithm was first implemented in the 1950s in custom hardware as one of the first artificial neural networks. Put simply, the algorithm visualized best resembles the “atom” of the neural network: a single, simple neuron that can answer decision problems (yes or no, black or white, etc). Perceptron maps a real-valued vector, x, to a binary output f(x) such that the dot product x dot w maps to either 1 or -1 (in the context of our data, class A or B). We iteratively update the weight vector based on if the output of the training set is correctly classified or misclassified. The parameter “eta” acts as a learning rate, controlling how much the weight vector changes with each misclassified data point.

.

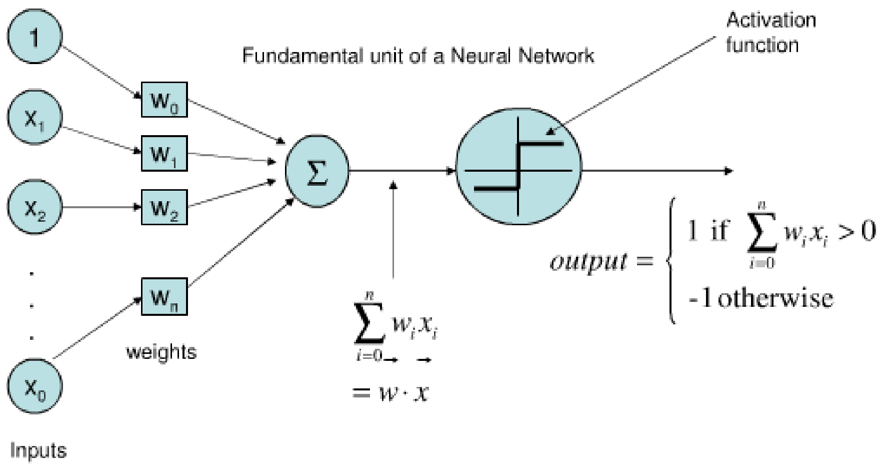

Fundamental unit of a Neural Network
Activation function
1
x_1
x_2
x_0
Inputs
w_0
w_1
w_2
w_n
weights
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure 1: Perceptron Animation

Our first step was to migrate a MATLAB implementation over to C to begin optimizing.

Serial code: This portion of the code initializes the array of weights and the array of misclassified flags. We set a flag not_classified so we know when to exit the main while loop.

```
int train_perceptron(data_t* x, char* y, double eta, int x_length, int x_dim) {
    double w[x_dim];
    double score[x_length];
    char misclassified[x_length];
    char not_classified = 1;
    int i, j, sum_missed, iters = 0;
    //set w to 0's, misclassified to 1's
    memset(w, 0, x_dim*sizeof(double));
    memset(misclassified, 1, x_length*sizeof(char));
}
```

Moving forward to the updating of weights within the while loop:

```
while(not_classified && iters <= MAX_ITERS){
        iters++;
        not_classified = 0;
        for(i=0; i < x_length; ++i){
            if(misclassified[i] == 1){
                for(j=0; j< x_dim; ++j){
                    w[j] = w[j] + eta*x[i*x_dim + j]*y[i];
                }
            }
        }
```

This step simply performs a check and update; if a data point is misclassified, we loop through the weight vector and error-correct it based on the above formula. In the serial implementation, this one double for loop updates all indices of the weight vector using all of the data.

Lastly, the classification step in the main loop:

```
 sum_missed = 0;
        for (i=0; i<x_length; ++i) {
            score[i] = 0;
            for (j = 0; j < x_dim; j++) {
                score[i] += x[i*x_dim + j]*w[j];
            }
            misclassified[i] = score[i]*y[i] <= 0.0 ? 1 : 0;
            // Set not_classified to 1 if any data point is misclassfied
            // and count number of missed.
            if (misclassified[i] == 1) {
                sum_missed++;
                not_classified = 1;
            }
        }
```

In this section of code, we take the dot product of x and w to determine what class our data point belongs to. If we are still misclassified, we increment sum_missed and set the flag not_classified to know that we need to continue on to another iteration.

We at first visualized Perceptron using a smaller data set which classifies data based on three different test cases.

Test Case 1 (Separated by a Line) - Code

```
case 1:
                y[i] = (0.2*(x[i*x_dim + 1] - 0.5)) +
                    (.6-x[i*x_dim + 2]) > 0 ? 1 : -1;
                break;
```

Test Case 2 (Separated by an Oval) - Code

```
 case 2:
                y[i] = (x[i*x_dim + 1]-.5)*(x[i*x_dim + 1]-.5) +
                    (x[i*x_dim + 2]-.5)*(x[i*x_dim + 2]-.5)
                    > 0.09 ? 1 : -1;
                break;
```

Test Case 3 (Separated by an Parabola) - Code

```
case 3:
                y[i] = 4*(x[i*x_dim + 1]-.5)*(x[i*x_dim + 1]-.5) +
                    (.2-x[i*x_dim + 2]) > 0 ? 1 : -1;
                break;
```

We performed several operations to preprocess our data before running the algorithm in order to achieve better separation results. Computing xy, $x * x$, and $y * y$ allows us to not only separate the 2 dimension data by line but also a parabola or ellipse.

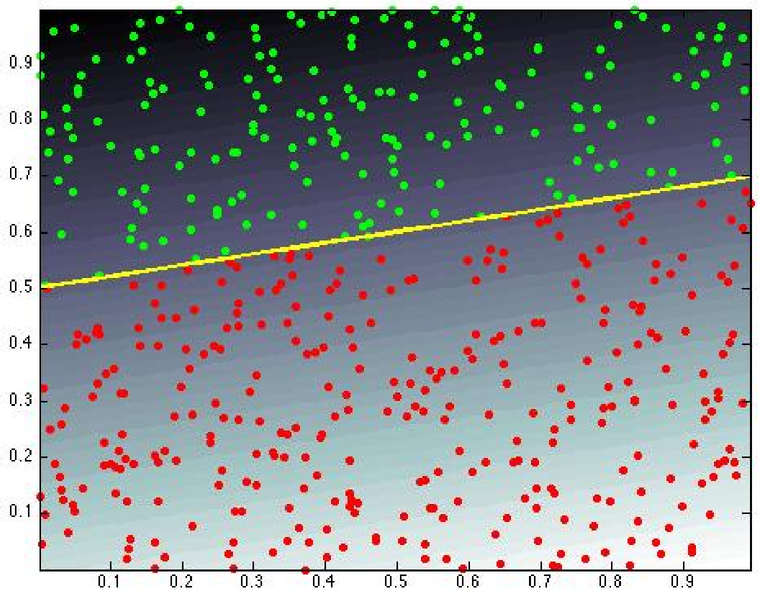


Figure 2: Test Case 1 Visualized

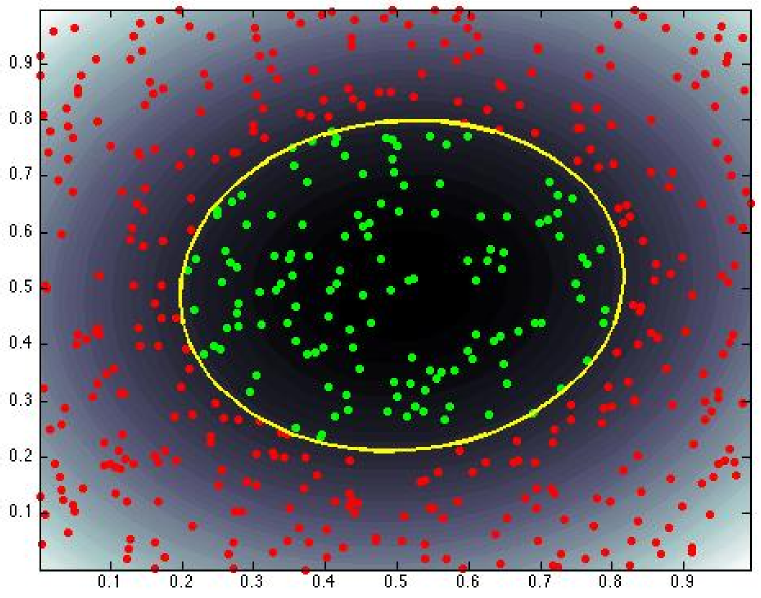


Figure 3: Test Case 2 Visualized

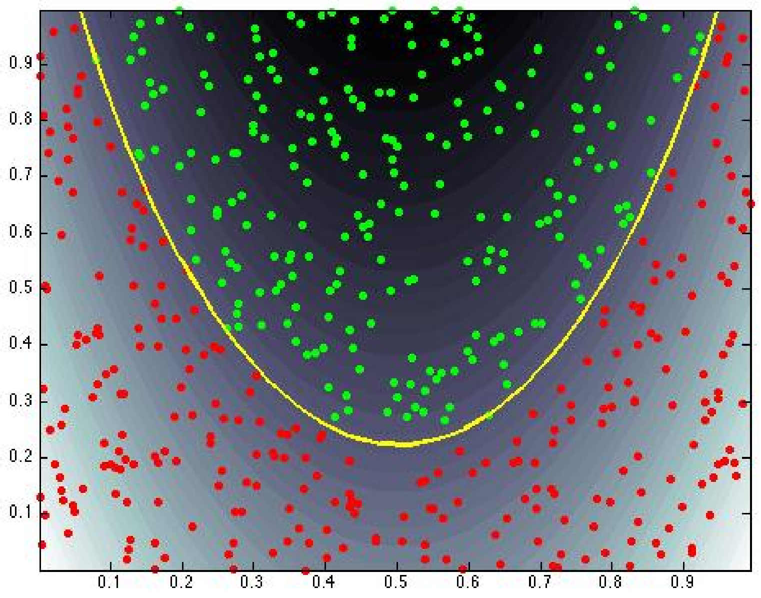
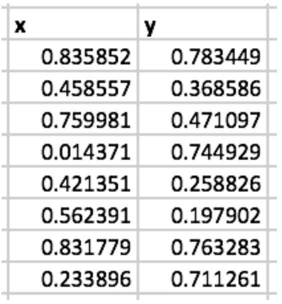
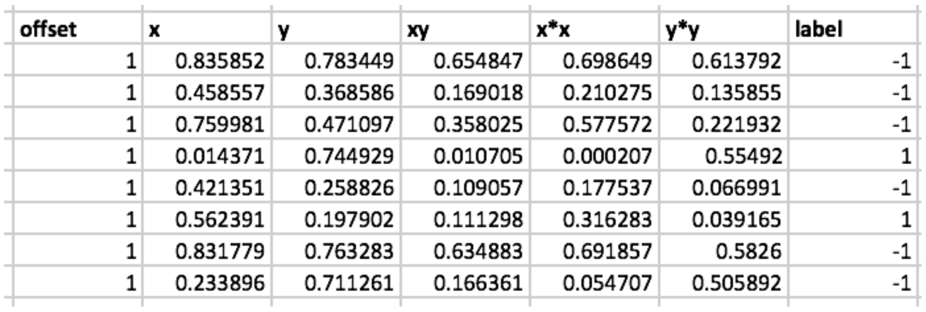


Figure 4: Test Case 3 Visualized



x	y
0.835852	0.783449
0.458557	0.368586
0.759981	0.471097
0.014371	0.744929
0.421351	0.258826
0.562391	0.197902
0.831779	0.763283
0.233896	0.711261

Figure 5: X, Y data generate randomly as values between 0 and 1.



offset	x	y	xy	x*x	y*y	label
1	0.835852	0.783449	0.654847	0.698649	0.613792	-1
1	0.458557	0.368586	0.169018	0.210275	0.135855	-1
1	0.759981	0.471097	0.358025	0.577572	0.221932	-1
1	0.014371	0.744929	0.010705	0.000207	0.55492	1
1	0.421351	0.258826	0.109057	0.177537	0.066991	-1
1	0.562391	0.197902	0.111298	0.316283	0.039165	1
1	0.831779	0.763283	0.634883	0.691857	0.5826	-1
1	0.233896	0.711261	0.166361	0.054707	0.505892	-1

Figure 6: Data after processing it to increase the dimensionailty and allow the perceptron algorithm to separate not just along a line.

3 Perceptron - Parallelization at a Glance

To parallelize the Perceptron, we could utilize either task or data parallelism. To elaborate, using task parallelism would involve having each thread operate on the data using a different "eta" value to better explore the space. In order to achieve a significant performance improvement for our test case, we needed to use data parallelism by having each thread operate and error correct on a portion of the weight vector in global memory. For this learning algorithm, we expect the accuracy of the results to decrease with datasets that are not completely separable within the provided dimensions and would require some transformation of the data to be able to separate the two classes with a single weight vector.

4 Perceptron - Multithreading with PThreads

The goal of this section of optimizing Perceptron was to utilize multi-core architecture in order to improve performance of the training phase of the perceptron algorithm. Each thread will operate on some number of indices of the global weight vector, and hence will require synchronization between threads. The question is whether or not the synchronization implementation is worth the overhead to parallelize the code. Multithreading the code requires the use of several global variables, mutexes, and barriers. Global variables include:

global_w - Global weight vector that will be updated at each iteration in the master loop. Each thread operates on some number of indices. For example, if we have eight threads and a dimensionality of 24, each thread will update three indices of the global weight vector.

global_not_classified - A flag that is shared between threads and checked each iteration by every thread order to ensure no threads escape the master loop before we finish classification.

global_sum_missed and global_countCheck - Counters used to assert that all threads finish classifying the data with no missed classifications.

We must set barriers when reading or writing to global_not_classified and global_sum_missed to ensure no stray threads get stuck in the main loop of the helper function.

Perceptron, much like SOR, requires synchronization between threads at every iteration of the main while loop. We must immediately wait for all threads to start the loop together, wait for all threads to finish updating the weight vector before we can begin re-classifying the data, and wait for all threads to finish changing the global variables (which represent critical sections of code). Hence, the multithreaded version of Perceptron requires a considerable amount of overhead and we expect that it may not reach a breakeven point with the serial implementation until the data set becomes quite large.

```
// each thread updates one index of the weight vector.
for(i = 0; i < X_length; ++i){
    if(misclassified[i] == 1){
        w += eta*X[i*(X_dim) + taskid]*y[i];
    }
}
// each thread updates global weight vector.
global_w[taskid] += w;
```

The code above shows the update step from the multithreaded perceptron implementation. We can see that each index of the weight vector is handled by a separate thread in parallel. For cases where the dimensionality exceeds the available number of CPU cores in the architecture, each thread may handle more than one index of the global weight vector. Here, when the number of misclassified items is small, the amount of work done by each thread will be very small also. At this point in the training, the overhead of synchronizing the threads makes the serial implementation perform better.

```
// each thread classifies 1/NUM_THREADS of the data.
sum_missed = 0;
not_classified = 0;
local_idx = 0;
for (i = X_length_low; i<X_length_high; ++i){
    score[local_idx] = 0;
    for(j = 0; j < X_dim; ++j){
        score[local_idx] += X[i*(X_dim) + j]*global_w[j];
    }

    misclassified[i] = score[local_idx]*y[i] <= 0.0 ? 1 : 0;
    // Set not_classified to 1 if any data point is misclassfied
    // and count number of missed.
    if (misclassified[i] == 1) {
        sum_missed++;
        not_classified = 1;
    }
    local_idx++;
}
```

In the classfication step, the training data is split between threads instead of splitting up the weight vector. This allows each thread to compute the new classifications of a 1/NUM_THREADS portion of the data, each doing so in parallel. Just as with the update step, when this thread finishes it must wait for all of the other threads to finish as well.

5 Perceptron - OpenCL and CUDA

In addition to the serial and pthread code, we also implemented Perceptron on the GPU for optimal speedup and parallelization. We implemented CUDA and OpenCL code to utilize the NVIDIA Tesla M2070 GPU device on scc1. While CUDA was developed by NVIDIA for parallel programming on GPUs, OpenCL is a framework that allows programs to run on heterogeneous devices. Because OpenCL is less specific to the GPU architecture, we expect the OpenCL execution time to be slower than the CUDA execution times, but faster than the CPU versions.

The CUDA and OpenCL implementations of Perceptron use two kernels: calculate_weights and classify. In both cases, the data is split amongst threads (CUDA) or work-items (OpenCL). For all data sizes, we used a set dimension of 300 threads/work-items per block/work-group. In the first kernel, each thread or work-item computes the weight vector for its assigned row of data if the data is misclassified. The weights are stored in a shared (CUDA) or local (OpenCL) vector, which is accumulated and copied into a weight matrix in the host. Within the host, the weight vectors from each block (CUDA) or work-group (OpenCL) are combined into an overall weight vector. The second kernel, classify, uses the overall weight vector to generate a score and determine whether each data point was correctly classified. Within the classify kernel, each thread or work-item calculates a score and updates the global misclassified and not_classified arrays.

To ensure synchronization and concurrency, we used several functions in both the kernel and host code. For CUDA, we used __syncthreads() to synchronize threads within a block, and cudaDeviceSynchronize() to synchronize the blocks in between kernel calls. In OpenCL, we used barrier(CLK_LOCAL_MEM_FENCE) to ensure all work-items "see" the same local data (i.e. to maintain concurrency). Within the host code, the clFinish() command prevents host code from executing until all work-groups have finished running the kernel.

To verify our all of our implementations for training the perceptron, we computed the dot product of the resulting weight matrix with the x vector and checked if the data was correctly classified or not. In all cases, the classifications matched and we were able to completely separate the data into the two fabricated classes (from the assigning labels manually).

6 Perceptron - Experiments and Results

We first tested the Perceptron on a small data set of 500 points and 6 dimensions (according to our preprocessing shown above) in order to verify that our code was working, and to graphically visualize our results. Moving on, we began testing our serial algorithm with -O3 optimization against three different implementations of Perceptron: multithreading with PThreads, and GPU implementations using CUDA and OpenCL. Our training set size ranged from 600 to 8000 data points. We modeled the execution time(s) as a function of the number of data points tested. As shown in Figure XXX, the serial version grows faster than the three parallel implementations, and appears to diverge from the bottom three curves. This is expected, as the classifications are very easily separable and require fewer iterations to converge to a solution. The speedups achieved for variable number of data points are as follows:

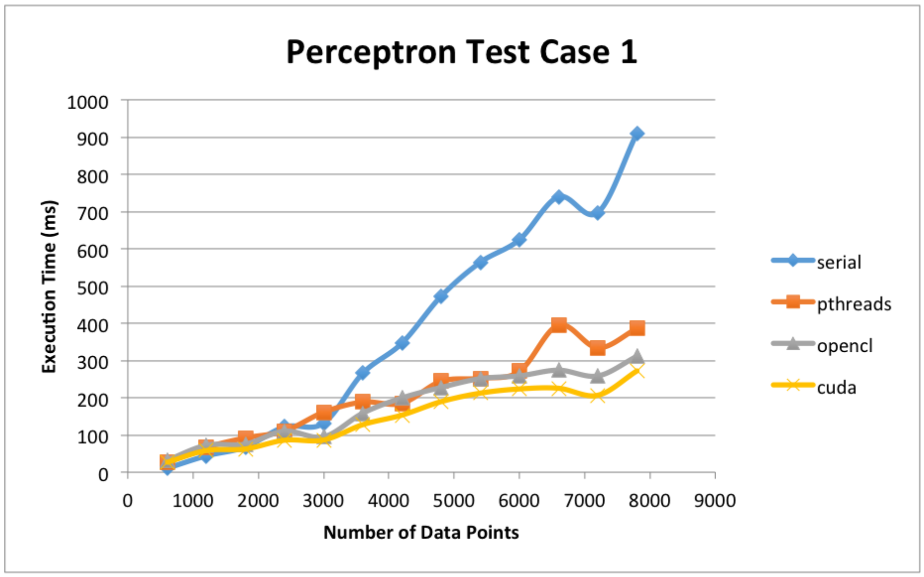


Figure 7: Perceptron Test Case 1

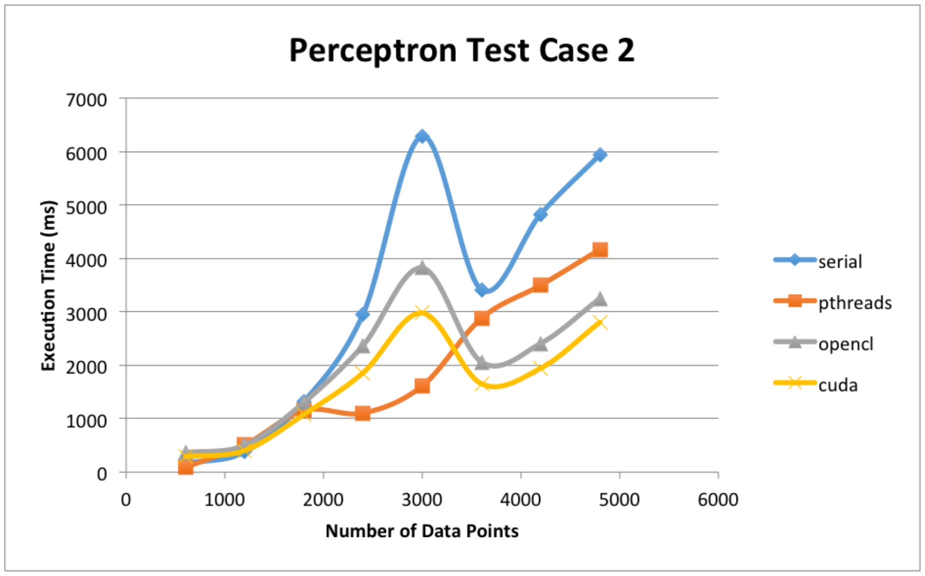


Figure 8: Perceptron Test Case 2

In this this test case (2), we separate the data by an oval. The number of iterations required to perfectly separate the data was on average significantly more than for the first test case. At data size 3000, the spike in the chart is due to an increased number of iterations required to perfectly separate the data (for these tests we did not look at convergence of the number of misclassified data points). The performance improvement for pthreads, OpenCL, and CUDA is not nearly as significant for this test case, because of the larger number of iterations. As the number of iterations grows, so does, generally speaking, the number of classified items. The overhead of synchronization between the parallel implementations then outweighs the benefits of separating the update and classification steps of the global weight vector. The alleviate this problem, we could simply end a pthread or GPU block?s existence when the number of misclassified elements it?s responsible for is small. However, making this modification occur at the right point in time for each thread would require quite a few tests to determine the overhead of running on N threads versus N/2 threads, for example. Furthermore, this would significantly increase the complexity of synchronization between threads and harder to achieve a verifiably correct result.

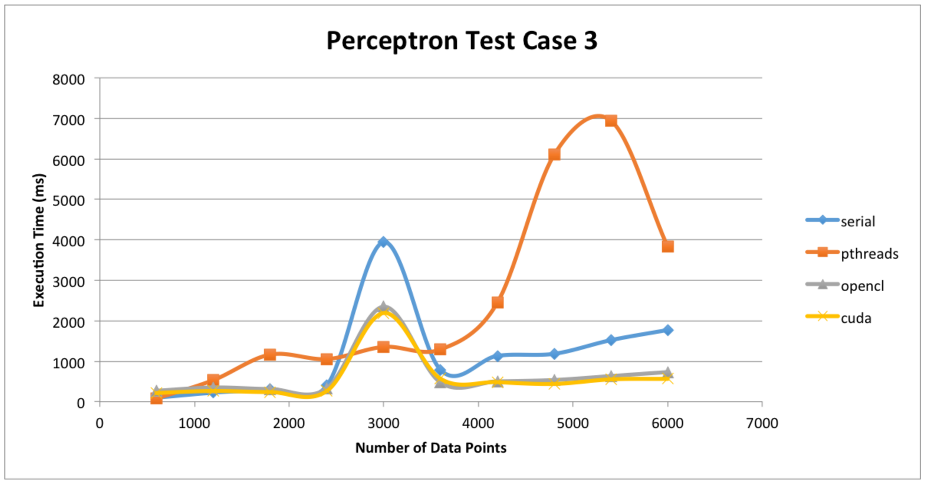


Figure 9: Perceptron Test Case 3

In this last test case (separated by a parabola), we see that for large data sizes the pthreads implementation takes longer than the serial version. The number of iterations required to get to perfect separation increases significantly for the pthreads implementation, but not at a data size of 3000, as it does with the serial and GPU implementations.

Perceptron: Average Speedup Over Serial Version			
	Test Case 1	Test Case 2	Test Case 3
PThreads	2.148	1.572	1.750
CUDA	2.587	1.943	1.961
OpenCL	1.841	1.683	0.459

7 K-Nearest Neighbors - An Overview

K-Nearest Neighbors is a lazy algorithm that classifies any new data point based on existing labeled data. A new point is compared to every labeled data point with some distance metric, for example euclidean or hamming distance. The K-nearest neighbors vote on the classification of the new data point by performing a mode operation. K-nearest neighbors, while unlike other machine learning algorithms in that the time it takes to classify a single data point grows with the number of training items, is still widely used to solve problems that are not well-understood or patterns are not intuitive. For example, K-nearest neighbors has been used to diagnose breast cancer from mammograms by comparing a new screening with a set of labeled mammograms. For KNN, we implemented a serial version of the code based on a MATLAB implementation and also a pthread implementation. We wanted to apply both KNN and perceptron to a real world dataset to compare running times and also accuracy for these two simple algorithms.

For KNN we used a dataset found on the UCI Machine Learning Repository, which has 23 attributes about each credit card holder and whether or not they defaulted on their next payment. It is an interesting application of machine learning, which may have practical relevance to a credit card issuer, that without spending a considerable amount of time understanding the various dimensions of the data is not immediately obvious that we would be able to achieve any accuracy at all. Still, we were able to apply KNN on the raw data from the dataset and achieve a classification rate of up to 78.5%. With Perceptron, we could not perfectly separate the raw data with a linear weight vector (this is where neural nets would come in), but we were able to converge to a correct classification rate of 78.8%. It is somewhat surprising that either algorithm would have any success at all on this data.

```
void perform_knn(const int k,
         const data_t* x,
         const data_t* labeled,
         const data_t* labels,
         const int dim,
         const int x_length,
         const int labeled_length,
         data_t* x_pred) {
    // Loop through data points and classify each based on nearest labeled
    // neighbors.
    for (int i = 0; i < x_length; i++) {
        if (i%100 == 0) {
            cout << "Done classifying: " << i << endl;
        }
        knn(k, x, i, labeled, labels, dim, labeled_length, x_pred);
    }
}
```

KNN is a very easy algorithm to parallelize because each classification is completely independent of the next one. However, each labeled data point is accessed as many times as there are unlabeled data points, so KNN has high computational intensity. With a GPU implementation is would make sense to separate the labeled data in blocks to take more advantage of caching.

```
void* knn_helper(void* threadarg){
    struct thread_data *my_data;
    my_data = (struct thread_data *) threadarg;
    int taskid = my_data->thread_id;
    int k = my_data->k;
    data_t* x = my_data->x;
    data_t* labeled = my_data->labeled;
    data_t* labels = my_data->labels;
    int dim = my_data->dim;
    int x_length = my_data->x_length;
    int labeled_length = my_data->labeled_length;
    data_t* x_pred = my_data->x_pred;
    int NUM_THREADS = my_data->NUM_THREADS;
    int x_length_low = (taskid * x_length)/NUM_THREADS;
    int x_length_high = x_length_low + (x_length/NUM_THREADS);

    printf("Hi! I am thread %d computing elements %d to %d\n",
            taskid,x_length_low,x_length_high);
    // Loop through data points and classify each based on nearest labeled
    // neighbors.
    for (int i = x_length_low; i < x_length_high; i++) {
        if (i%100 == 0) {
            cout << "Done classifying: " << i << endl;
        }
        knn(k, x, i, labeled, labels, dim, labeled_length, x_pred);
    }
}
```

Here we see the naive but effective parallel implementation of KNN. Because each classification is independent, it is straightforward to simply divide the data up into equal parts and process the unlabeled data in parallel with as many threads as are available. For the dataset we used of 25000 labeled points, dividing up the data in any other way would not provide significant cache benefits unfortunately. If instead we divided up the labeled data and synchronized the threads at the end of each classification of a single data point, each thread would still need to operate on 2500 data points of 23 doubles, a size that exceeds the available cache per thread of our CPU.

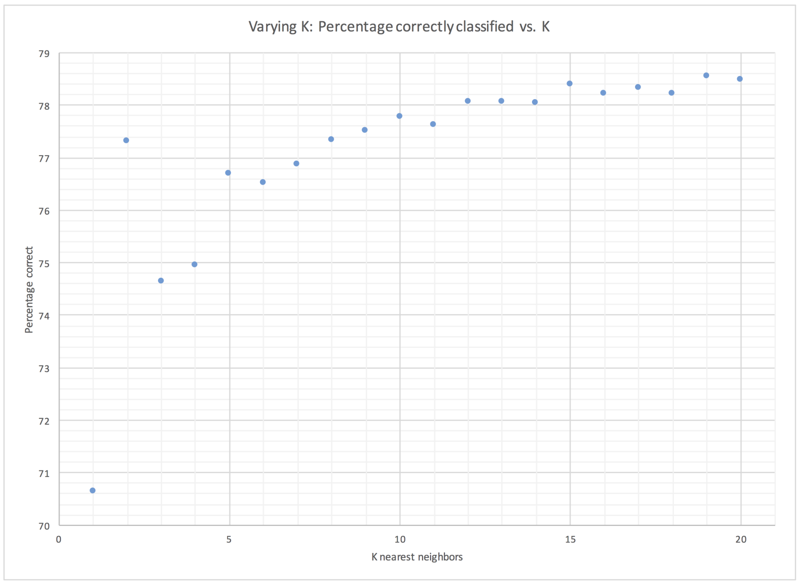


Figure 10: Varying K and Tracking ClassifiedPercentage

This chart is interesting because it shows how we can, generally speaking, improve the performance of KNN by increasing the number of neighbors that are considered, especially for an application like this, where applying a euclidean distance between various pieces of information of two credit card holders does not intuitively make very much sense. We can see that as we increase the number of nearest neighbors considered from 1 to 20 the classification rate increases from about 70.5% to 78.5% (but it starts to plateau). However, we were more concerned about the timing implications of increasing K for the serial and parallel versions and will explore them below.

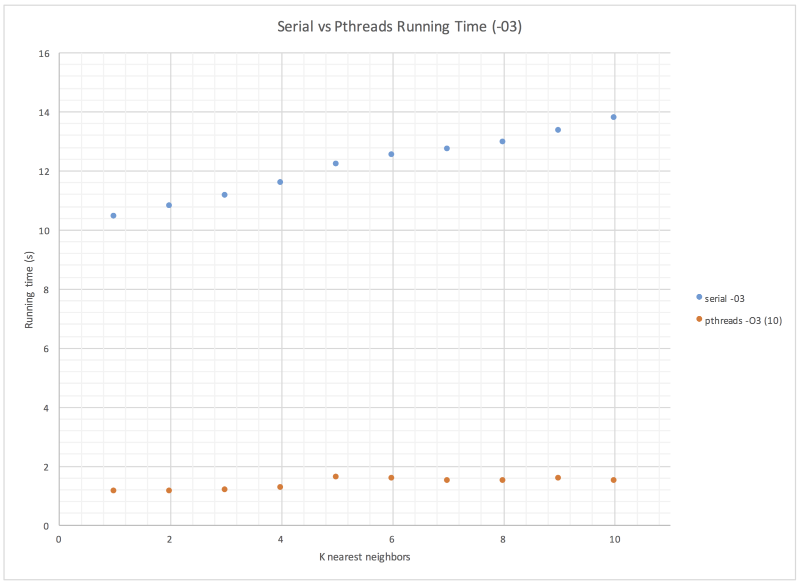


Figure 11: KNN: Serial vs PThreads Running Time (-O3)

In this chart we compare the running time of serial KNN versus our pthreads implementation running 10 threads with -O3 optimizations. We are looking at the total running time for both to classify 5000 data points using 25000 labeled points. Over these ten sizes for K, we saw an average speedup of 8.7 for the parallel implementation.

Perceptron actually performs far better on this dataset. We have implemented code that checks for convergence by saving a value which stores the minimum number missed accross all iterations. If this minimum is not surpassed after a set number of iterations, the perceptron algorithm stops.

```
        if (min_missed == -1 || sum_missed < min_missed) {
            min_missed = sum_missed;
            iters_since_best = 0;
            for (j = 0; j < x_dim; j++) {
                w_converge[j] = w[j];
            }
        } else {
            if (iters_since_best >= 1000) {
                for (j = 0; j < x_dim; j++) {
                    w[j] = w_converge[j];
                }
                break;
            }
            iters_since_best++;
        }
```

Using this approach with 1000 iterations of no improvement as our threshold, we were able to classify 19434 of the 25000 training data points. Then, during testing we saw a classification rate of 78.8% (3942 out of 5000). To find this weight vector took 1814 iterations total, so the best weight vector occurred at iteration 814 (followed by 1000 iterations of no improvement). Using our serial implementation of the perceptron algorithm, compiled with -O3 optimizations, we see a running of time of 2.1 seconds, significantly better than the running times of serial KNN and with a slightly better classification rate. This real world dataset shows that picking and tuning the right machine learning algorithm is going to be the most important thing to do to improve performance.

8 Introduction