Daniel Dias
Edmond Chow
CX 4220
3 May 2014

Netflix Recommender Using Stochastic Gradient Descent

Overview:

The Netflix recommending system is an intricate piece of software designed to match users with movies that he/she would be interested in. Recommender systems exist in many forms and there are many different ways to implement this type of system. This implementation of the Netflix recommender system only matches users with movies based on previous movies that the users have previously rated themselves.

More specifically to this particular implementation, the problem solved is a minimization problem where we are trying to minimize the error between predicted values and given data; however, noise is also accounted for, so it becomes a regularized minimization problem of the form

$$\min_{q^*,p^*} \sum_{(u,i) \in \kappa} (r_{ui} - q_i \bullet p_u)^2 + \lambda(|q_i|^2 + |p_u|^2).$$

This problem is solved using a parallelized stochastic gradient descent algorithm using a lock-free implementation that relied on the data being sufficiently sparse that items did not overwrite each other very often.

The stochastic gradient descent algorithm was as of the form, in pseudo-code,

```
Loop until convergence
{
    Parallel Loop over learning data
    {
        q_i = q_i + γ(e_ui p_u – λq_i)
        p_u = p_u + γ(e_ui q_i – λp_u)
    }
}
```

where $e_{ui} = r_{ui} - q_i \bullet p_u$, and q_i, p_u are vectors of length *f*.

Sequential Implementation:

A sequential version of this implementation is used as a baseline for timings. Convergence is tested in 2 ways, if the aggregate squared error over the entire iteration is below a certain threshold, or if that value stops decreasing but instead just stagnates. Also, the baseline is for the entire 99,072,112 entries of the original learning data, and running that much data sequentially will take too long to complete, so the baseline will be determined by timing a fixed number of iterations. The aggregate error equals

$$\sqrt{((1 / N) * \sum(r_{ui} - q_i \bullet p_u)^2)},$$

where N is the length of the training data.

Because of the way the GPU schedules blocks and due to its unpredictable nature, adding to this single value while updating p's and q's generates unpredictable results, so this aggregate error is calculated on its own traversal of the training data set. The sequential implementation also only calculates this aggregate error on a second pass per iteration to mimic the GPU if it were a sequential implementation.

The time required to run 10 iterations sequentially is about 965.143818 seconds (16 minutes, 5.143818 seconds).

Parallel Implementation:

Data partitioning was fairly straight forward and they were as follows: the learning data set is a sparse matrix and is laid out in memory as a long array of 99,072,112 rating_t structs each with 3 integers named userIndex, movieIndex, and rating; the user factors matrix consists of the number of users p_u vectors, each of length *f*, and the movie factors matrix which is a matrix consisting of number of movies q_i vectors, each also of length *f*. Both of the factor matrices are single arrays of doubles allocated as 1 dimensional arrays. In short:

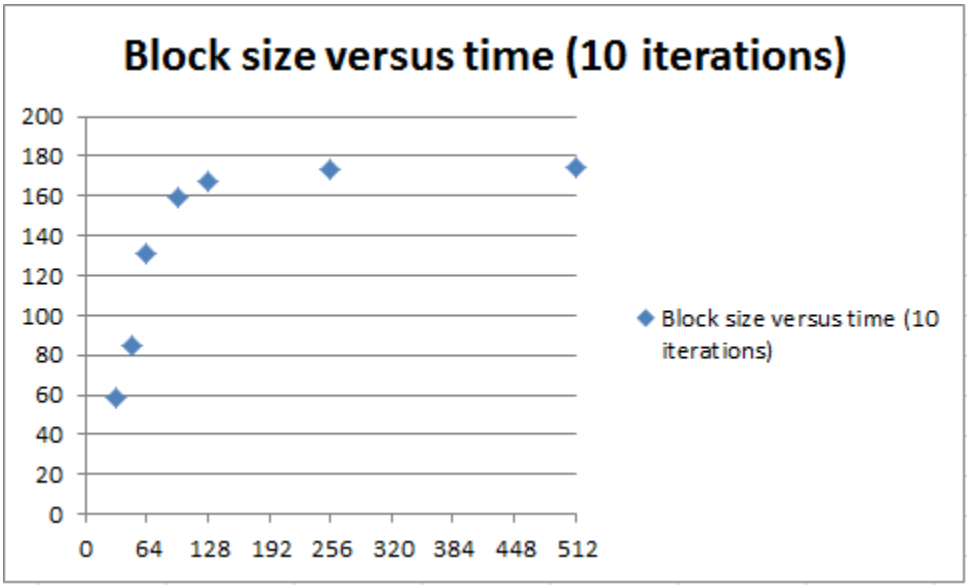
```
rating_t* learningData[LEARNING_DATA_SIZE];
double* userFactors[numberUsers * FACTORS];
double* movieFactors[numberMovies * FACTORS];
```

The learning data was also scrambled before the algorithm began because the file that carries the learning data has every movie together. Leaving them together would leave a very high probably that movie factors would get overwritten due to data racing conditions. Items in the rating_t array were scrambled by looping through LEARNING_DATA_SIZE divided by number of movies and randomly going to another section of the array and swapping the two around.

The learning data was split using a 1 dimensional partition among blocks; however, the GPU cannot handle a grid width of (99,072,112 – 1) / BLOCKSIZE + 1, so the blocks had to be placed on a 2 dimensional grid where the indexing makes it behave like it's 1 dimensional again. Indexing was done using the built in variables used for block and thread indexing. Since every single bit of required data is allocated to the GPU's memory, every block has access to all of the data; however, the parallelism is done by having each thread have its own index, and then each thread use a different learning data rating, user, movie pair. Indexing is done as follows

```
int index = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x + threadIdx.x;
```

Timings for different block sizes:



Block Size	Seconds (10 iterations)
32	58.321979
48	84.699867
64	130.887128
96	158.884226
128	167.808483
256	173.038329
512	174.833201

Sequential 965.143818 seconds

The speedup between the sequential algorithm and the parallel implementation using a block size of 32 is 965.143818 divided by 58.321979 creates a speedup of 16.549. This amount of speedup makes sense for a few reasons. Amdahl's law states that if the problem does not increase in size, the speedup is bounded by the inverse of the sequential portion. For the part that is parallelizable, the GPUs that the Jinx cluster has, the M2090, carries a lot of cores that were able to constantly keep working, especially if the block size was small. Also, each iteration of my kernel code only required 20 registers.

Results:

User#:427501	Movie#:8	Predicted Rating = 3.171480
User#:260749	Movie#:18	Predicted Rating = 3.067682
User#:311872	Movie#:28	Predicted Rating = 3.832538
User#:73318	Movie#:30	Predicted Rating = 3.177767
User#:182071	Movie#:30	Predicted Rating = 2.864315

Improvements:

One improvement that really sticks out is the fact that my code calculates the error term for a particular learning data entry, updates the associated p or q vector, and then does not add this value to the aggregate error. After traversing all of the learning data that first time, it does it again, but this time only to calculate the aggregate error. The reason that this is implemented this way is that if a single iteration at a learning data entry added the error term to the aggregate error and then updated the p and q vectors, it would write these new p and q vectors straight back to DRAM, and future blocks over proceeding learning data entries would use updated values that had been multiplied by the regularization parameter, λ, and the learning rate, γ, making the aggregate error likely to overflow. This issue did not arise with $\lambda = 0.005$ and $\gamma = 0.00125$ as these values kept the updated values and the old values in balance.

A way to fix this issue would be to update the aggregate error term and the p & q vectors in the same iteration, but have the output written to a second buffer where the output would just write to the buffer that is not being read from like a toggle. This fix would require twice as much memory for p and q, which are both dense matrices.

Appendix

-- Makefile

```
naive:
	gcc -g -std=c99 -fopenmp netflix_naive.c -o Netflix -lm

nvcc:
	nvcc -arch=sm_20 netflix.cu -o Netflix -Xcompiler -fopenmp -lgomp

nvcc_ptxas:
	nvcc -arch=sm_20 netflix.cu -o Netflix -Xcompiler -fopenmp -lgomp --ptxas-options=-v

counter:
	gcc learning_data_counter.c -o Counter
	./Counter

clear:
	rm Counter
	rm Netflix
```

```
-- netflix.cu
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/time.h>

const int FACTORS = 30;

int numberMovies = 0;
int numberUsers = 0;

#define LEARNING_RATE (0.00125)
#define REGULARIZATION_PARAMETER (0.005)

#define BLOCKSIZE (32)
#define LEARNING_DATA_SIZE (99072112)
#define ERROR_CHANGE_THRESHOLD (100)
#define ERROR_THRESHOLD (0.01)//(0.145)
#define WARP_SIZE (32)

#define gpuErrorCheck(x) { gpuAssert((x), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char* fileName, int lineNumber, int abort = 1)
{
	if (code != cudaSuccess)
	{
		fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), fileName,
lineNumber);
		if (abort)
			exit(code);
	}
}

typedef struct rating
{
	int rating;
	int userIndex;
	int movieIndex;
} rating_t;

rating_t* h_learningData;
```

```
double* h_userFactors;
double* h_movieFactors;

double get_walltime()
{
	struct timeval tp;
	gettimeofday(&tp, NULL);
	return (double)(tp.tv_sec + tp.tv_usec*1e-6);
}

void CreateLearningDataAndFactors()
{
	h_learningData = (rating_t*)malloc(sizeof(rating_t) * LEARNING_DATA_SIZE);
	assert(NULL != h_learningData);

	int indexLearningData = 0;
	FILE* netflixDataFile = fopen("netflix.dat", "r");
	char buffer[60];
	while (indexLearningData < LEARNING_DATA_SIZE && fgets(buffer, 60,
netflixDataFile) != NULL)
	{
		assert(indexLearningData < LEARNING_DATA_SIZE);

		char* startPointer = buffer;
		char* endPointer = buffer;

		h_learningData[indexLearningData].userIndex = ((int)strtol(startPointer,
&endPointer, 10)) - 1;
		startPointer = endPointer;
		h_learningData[indexLearningData].movieIndex = ((int)strtol(startPointer,
&endPointer, 10)) - 1;
		startPointer = endPointer;
		h_learningData[indexLearningData].rating = ((int)strtol(startPointer,
&endPointer, 10));

		if (h_learningData[indexLearningData].userIndex > numberUsers)
			numberUsers = h_learningData[indexLearningData].userIndex;
		if (h_learningData[indexLearningData].movieIndex > numberMovies)
			numberMovies = h_learningData[indexLearningData].movieIndex;
```

```
            ++indexLearningData;
        }
        ++numberMovies;
        ++numberUsers;

        printf("Allocating memory for user and movie factors\n");

        h_userFactors = (double*)malloc(sizeof(double) * FACTORS * numberUsers);
        assert(NULL != h_userFactors);
        h_movieFactors = (double*)malloc(sizeof(double) * FACTORS * numberMovies);
        assert(NULL != h_movieFactors);

        #pragma omp parallel for
        for (int n = 0; n < (FACTORS * numberUsers); ++n)
            h_userFactors[n] = 1.0;

        int index = 0;
        int currentMovieIndex = 0;
        size_t sumRating = 0;
        size_t timesAdded = 0;
        while (index < LEARNING_DATA_SIZE)
        {
            if (h_learningData[index].movieIndex != currentMovieIndex)
            {
                for (int f = 0; f < FACTORS; ++f)
                    h_movieFactors[currentMovieIndex * FACTORS + f] =
(sumRating / ((double)timesAdded)) / ((double)FACTORS);

                currentMovieIndex = h_learningData[index].movieIndex;
                sumRating = 0;
                timesAdded = 0;
                continue;
            }

            sumRating += h_learningData[index].rating;
            ++timesAdded;
            ++index;
        }
        for (int f = 0; f < FACTORS; ++f)
```

```
                h_movieFactors[currentMovieIndex * FACTORS + f] = (sumRating / ((double)
timesAdded)) / ((double)FACTORS);

        printf("Randomizing the learning data\n");

        srand(time(NULL));

        for (int n = 0; n < (LEARNING_DATA_SIZE / numberMovies); ++n)
        {
                rating_t tempRatingBlock;
                size_t m = ((rand() % numberMovies) * (LEARNING_DATA_SIZE /
numberMovies)) + (rand() % (LEARNING_DATA_SIZE / numberMovies));
                assert(m < LEARNING_DATA_SIZE);

                tempRatingBlock.rating = h_learningData[n].rating;
                tempRatingBlock.userIndex = h_learningData[n].userIndex;
                tempRatingBlock.movieIndex = h_learningData[n].movieIndex;

                h_learningData[n].rating = h_learningData[m].rating;
                h_learningData[n].userIndex = h_learningData[m].userIndex;
                h_learningData[n].movieIndex = h_learningData[m].movieIndex;

                h_learningData[m].rating = tempRatingBlock.rating;
                h_learningData[m].userIndex = tempRatingBlock.userIndex;
                h_learningData[m].movieIndex = tempRatingBlock.movieIndex;
        }
}

__host__ __device__ double DotProduct(const double* vectorA, const double* vectorB, const
int length)
{
        double dotProduct = 0.0;

        for (int n = 0; n < length; ++n)
                dotProduct += vectorA[n] * vectorB[n];

        return dotProduct;
}
```

```
__global__ void SingleIteration(const int learningDataLength, double* sumErrorSquared,
rating_t* learningData, double* userFactors, double* movieFactors)
{
	//int index_x = blockIdx.x * blockDim.x + threadIdx.x;
	//int index_y = blockIdx.y * blockDim.y + threadIdx.y;
	//int index = index_y * gridDim.x * blockDim.x + index_x;
	int indexKnownData = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
+ threadIdx.x;
	if (indexKnownData == 0)
		*sumErrorSquared = 0.0;
	if (indexKnownData < learningDataLength)
	{
		int userIndex = learningData[indexKnownData].userIndex;
		int movieIndex = learningData[indexKnownData].movieIndex;
		int rating = learningData[indexKnownData].rating;

		double errorTerm = rating - DotProduct(&movieFactors[movieIndex *
FACTORS], &userFactors[userIndex * FACTORS], FACTORS);
		//*sumErrorSquared += (errorTerm * errorTerm);

		for (int f = 0; f < FACTORS; ++f)
		{
			movieFactors[movieIndex * FACTORS + f] += LEARNING_RATE *
(errorTerm * userFactors[userIndex * FACTORS + f]
				- REGULARIZATION_PARAMETER *
movieFactors[movieIndex * FACTORS + f]);
		}

		for (int f = 0; f < FACTORS; ++f)
		{
			userFactors[userIndex * FACTORS + f] += LEARNING_RATE *
(errorTerm * movieFactors[movieIndex * FACTORS + f]
				- REGULARIZATION_PARAMETER * userFactors[userIndex *
FACTORS + f]);
		}
	}
}

__global__ void CalculateAggregateError(const int learningDataLength, double*
sumErrorSquared, rating_t* learningData, double* userFactors, double* movieFactors)
```

```
{
	int indexKnownData = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
+ threadIdx.x;
	if (indexKnownData < learningDataLength)
	{
		int userIndex = learningData[indexKnownData].userIndex;
		int movieIndex = learningData[indexKnownData].movieIndex;
		int rating = learningData[indexKnownData].rating;

		double errorTerm = rating - DotProduct(&movieFactors[movieIndex *
FACTORS], &userFactors[userIndex * FACTORS], FACTORS);
		*sumErrorSquared += (errorTerm * errorTerm);
	}
}

void PrintPredictedData()
{
	int indexQueryData = 0;
	FILE* netflixQueryFile = fopen("netflix_query", "r");
	char buffer[60];
	while (fgets(buffer, 60, netflixQueryFile) != NULL)
	{
		char* startPointer = buffer;
		char* endPointer = buffer;

		int userIndex = ((int)strtol(startPointer, &endPointer, 10));
		startPointer = endPointer;
		int movieIndex = ((int)strtol(startPointer, &endPointer, 10));
		startPointer = endPointer;

		assert(userIndex < numberUsers);
		assert(movieIndex < numberMovies);

		double predictedRating = DotProduct(&h_movieFactors[movieIndex *
FACTORS], &h_userFactors[userIndex * FACTORS], FACTORS);

		printf("User#:%d\tMovie#:%d\tPredicted Rating = %lf\n", userIndex,
movieIndex, predictedRating);

		++indexQueryData;
```

```
        }
}

int main(int argc, char** argv)
{
        printf("Starting Netflix Recommender System Algorithm with Stochastic Gradient
Descent\n");
        printf("Creating Learning Data\n");

        CreateLearningDataAndFactors();

        printf("Copying Data to GPU\n");

        double* h_sumErrorSquared = (double*)malloc(sizeof(double));
        double* d_sumErrorSquared;
        *h_sumErrorSquared = 0.0;
        gpuErrorCheck(cudaMalloc((void**)&d_sumErrorSquared, sizeof(double)));
        gpuErrorCheck(cudaMemcpy(d_sumErrorSquared, h_sumErrorSquared, sizeof(double),
cudaMemcpyHostToDevice));

        double* d_userFactors;
        double* d_movieFactors;
        rating_t* d_learningData;

        gpuErrorCheck(cudaMalloc((void**)&d_userFactors, sizeof(double) * FACTORS *
numberUsers));
        gpuErrorCheck(cudaMalloc((void**)&d_movieFactors, sizeof(double) * FACTORS *
numberMovies));
        gpuErrorCheck(cudaMalloc((void**)&d_learningData, sizeof(rating_t) *
LEARNING_DATA_SIZE));

        gpuErrorCheck(cudaMemcpy(d_userFactors, h_userFactors, sizeof(double) * FACTORS
* numberUsers, cudaMemcpyHostToDevice));
        gpuErrorCheck(cudaMemcpy(d_movieFactors, h_movieFactors, sizeof(double) *
FACTORS * numberMovies, cudaMemcpyHostToDevice));
        gpuErrorCheck(cudaMemcpy(d_learningData, h_learningData, sizeof(rating_t) *
LEARNING_DATA_SIZE, cudaMemcpyHostToDevice));

        printf("Starting Algorith now\n");
        fflush(stdout);
```

```
	dim3 blockSize;
	blockSize.x = BLOCKSIZE;

	dim3 gridSize;

	int numberBlocksRequired = ((LEARNING_DATA_SIZE - 1) / BLOCKSIZE) + 1;

	if (numberBlocksRequired > 65535)
	{
		gridSize.x = 65535;
		gridSize.y = (numberBlocksRequired - 1) / 65535 + 1;
	}
	else
	{
		gridSize.x = numberBlocksRequired;
		gridSize.y = 1;
	}

	double previousErrorSquared = 9999999;
	double previousPreviousErrorSquared;

	unsigned int iterations = 0;
	double startTime = get_walltime();
	do
	{
		previousPreviousErrorSquared = previousErrorSquared;
		previousErrorSquared = *h_sumErrorSquared;

		SingleIteration<<<gridSize, blockSize>>>(LEARNING_DATA_SIZE,
d_sumErrorSquared, d_learningData, d_userFactors, d_movieFactors);
		gpuErrorCheck(cudaPeekAtLastError());
		gpuErrorCheck(cudaDeviceSynchronize());

		CalculateAggregateError<<<gridSize, blockSize>>>(LEARNING_DATA_SIZE,
d_sumErrorSquared, d_learningData, d_userFactors, d_movieFactors);
		gpuErrorCheck(cudaPeekAtLastError());
		gpuErrorCheck(cudaDeviceSynchronize());
```

```
		gpuErrorCheck(cudaMemcpy(h_sumErrorSquared, d_sumErrorSquared, sizeof(double), cudaMemcpyDeviceToHost));

		printf("\th_sumErrorSquared = %lf, (1/N(h_sumErrorSquared * WARP_SIZE)) ^(1/2) = %lf\n", *h_sumErrorSquared, sqrt(1.0 / LEARNING_DATA_SIZE * (*h_sumErrorSquared * WARP_SIZE)));
		++iterations;

		if (*h_sumErrorSquared > previousErrorSquared && previousErrorSquared > previousPreviousErrorSquared ||
			(abs(previousErrorSquared - *h_sumErrorSquared) + abs(previousPreviousErrorSquared - previousErrorSquared)) / 2.0 < ERROR_CHANGE_THRESHOLD)
			break;
	} while (*h_sumErrorSquared > (double)LEARNING_DATA_SIZE / (double) WARP_SIZE * (ERROR_THRESHOLD * ERROR_THRESHOLD));
	double endTime = get_walltime();

	printf("Performed %u iterations\n", iterations);

	printf("Total time = %lf\n", endTime - startTime);
	printf("Copying user and movie factors from GPU to host\n");

	gpuErrorCheck(cudaMemcpy(h_userFactors, d_userFactors, sizeof(double) * FACTORS * numberUsers, cudaMemcpyDeviceToHost));
	gpuErrorCheck(cudaMemcpy(h_movieFactors, d_movieFactors, sizeof(double) * FACTORS * numberMovies, cudaMemcpyDeviceToHost));

	PrintPredictedData();

	gpuErrorCheck(cudaFree(d_sumErrorSquared));
	gpuErrorCheck(cudaFree(d_userFactors));
	gpuErrorCheck(cudaFree(d_movieFactors));
	gpuErrorCheck(cudaFree(d_learningData));

	free(h_sumErrorSquared);
	free(h_userFactors);
	free(h_movieFactors);
	free(h_learningData);
```

```
        return 0;
}
```

```
 -- netflix_naive.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>

const int FACTORS = 30;

int numberMovies = 0;
int numberUsers = 0;

#define LEARNING_RATE (0.00125)
#define REGULARIZATION_PARAMETER (0.005)

#define BLOCKSIZE (256)
#define LEARNING_DATA_SIZE (99072112)
#define ERROR_THRESHOLD (0.145)

typedef struct rating
{
        int rating;
        int userIndex;
        int movieIndex;
} rating_t;

rating_t* h_learningData;

double* h_userFactors;
double* h_movieFactors;

double get_walltime()
{
        struct timeval tp;
        gettimeofday(&tp, NULL);
        return (double)(tp.tv_sec + tp.tv_usec*1e-6);
}

void CreateLearningDataAndFactors()
```

```
{
	h_learningData = (rating_t*)malloc(sizeof(rating_t) * LEARNING_DATA_SIZE);
	assert(NULL != h_learningData);

	int indexLearningData = 0;
	FILE* netflixDataFile = fopen("netflix.dat", "r");
	char buffer[60];
	while (indexLearningData < LEARNING_DATA_SIZE && fgets(buffer, 60,
netflixDataFile) != NULL)
	{
		assert(indexLearningData < LEARNING_DATA_SIZE);

		char* startPointer = buffer;
		char* endPointer = buffer;

		h_learningData[indexLearningData].userIndex = ((int)strtol(startPointer,
&endPointer, 10)) - 1;
		startPointer = endPointer;
		h_learningData[indexLearningData].movieIndex = ((int)strtol(startPointer,
&endPointer, 10)) - 1;
		startPointer = endPointer;
		h_learningData[indexLearningData].rating = ((int)strtol(startPointer,
&endPointer, 10));

		if (h_learningData[indexLearningData].userIndex > numberUsers)
			numberUsers = h_learningData[indexLearningData].userIndex;
		if (h_learningData[indexLearningData].movieIndex > numberMovies)
			numberMovies = h_learningData[indexLearningData].movieIndex;

		++indexLearningData;
	}
	++numberMovies;
	++numberUsers;

	printf("Allocating memory for user and movie factors\n");

	h_userFactors = (double*)malloc(sizeof(double) * FACTORS * numberUsers);
	assert(NULL != h_userFactors);
	h_movieFactors = (double*)malloc(sizeof(double) * FACTORS * numberMovies);
	assert(NULL != h_movieFactors);
```

```
	#pragma omp parallel for
	for (int n = 0; n < (FACTORS * numberUsers); ++n)
		h_userFactors[n] = 1.0;

	int index = 0;
	int currentMovieIndex = 0;
	size_t sumRating = 0;
	size_t timesAdded = 0;
	while (index < LEARNING_DATA_SIZE)
	{
		if (h_learningData[index].movieIndex != currentMovieIndex)
		{
			for (int f = 0; f < FACTORS; ++f)
				h_movieFactors[currentMovieIndex * FACTORS + f] =
(sumRating / ((double)timesAdded)) / ((double)FACTORS);

			currentMovieIndex = h_learningData[index].movieIndex;
			sumRating = 0;
			timesAdded = 0;
			continue;
		}

		sumRating += h_learningData[index].rating;
		++timesAdded;
		++index;
	}
	for (int f = 0; f < FACTORS; ++f)
		h_movieFactors[currentMovieIndex * FACTORS + f] = (sumRating / ((double)
timesAdded)) / ((double)FACTORS);

	printf("Randomizing the learning data\n");

	srand(time(NULL));

	for (int n = 0; n < (LEARNING_DATA_SIZE / numberMovies); ++n)
	{
		rating_t tempRatingBlock;
		size_t m = ((rand() % numberMovies) * (LEARNING_DATA_SIZE /
numberMovies)) + (rand() % (LEARNING_DATA_SIZE / numberMovies));
```

```
                assert(m < LEARNING_DATA_SIZE);

                tempRatingBlock.rating = h_learningData[n].rating;
                tempRatingBlock.userIndex = h_learningData[n].userIndex;
                tempRatingBlock.movieIndex = h_learningData[n].movieIndex;

                h_learningData[n].rating = h_learningData[m].rating;
                h_learningData[n].userIndex = h_learningData[m].userIndex;
                h_learningData[n].movieIndex = h_learningData[m].movieIndex;

                h_learningData[m].rating = tempRatingBlock.rating;
                h_learningData[m].userIndex = tempRatingBlock.userIndex;
                h_learningData[m].movieIndex = tempRatingBlock.movieIndex;
        }
}

double DotProduct(const double* vectorA, const double* vectorB, const int length)
{
        double dotProduct = 0.0;

        for (int n = 0; n < length; ++n)
                dotProduct += vectorA[n] * vectorB[n];

        return dotProduct;
}

void PrintPredictedData()
{
        int indexQueryData = 0;
        FILE* netflixQueryFile = fopen("netflix_query", "r");
        char buffer[60];
        while (fgets(buffer, 60, netflixQueryFile) != NULL)
        {
                char* startPointer = buffer;
                char* endPointer = buffer;

                int userIndex = ((int)strtol(startPointer, &endPointer, 10));
                startPointer = endPointer;
                int movieIndex = ((int)strtol(startPointer, &endPointer, 10));
                startPointer = endPointer;
```

```
			assert(userIndex < numberUsers);
			assert(movieIndex < numberMovies);

			double predictedRating = DotProduct(&h_movieFactors[movieIndex *
FACTORS], &h_userFactors[userIndex * FACTORS], FACTORS);

			printf("User#:%d\tMovie#:%d\tPredicted Rating = %lf\n", userIndex,
movieIndex, predictedRating);

			++indexQueryData;
		}
}

int main(int argc, char** argv)
{
		printf("Starting Netflix Recommender System Algorithm with Stochastic Gradient
Descent\n");
		printf("Creating Learning Data\n");

		CreateLearningDataAndFactors();

		double* h_sumErrorSquared = (double*)malloc(sizeof(double));
		*h_sumErrorSquared = 0.0;

		printf("Starting Algorith now\n");
		fflush(stdout);

		unsigned int iterations = 0;
		double startTime = get_walltime();
		do
		{
			for (int indexKnownData = 0; indexKnownData < LEARNING_DATA_SIZE;
++indexKnownData)
			{
				if (indexKnownData == 0)
					*h_sumErrorSquared = 0.0;
				if (indexKnownData < LEARNING_DATA_SIZE)
				{
					int userIndex = h_learningData[indexKnownData].userIndex;
```

```
                                int movieIndex = h_learningData[indexKnownData].movieIndex;
                                int rating = h_learningData[indexKnownData].rating;

                                double errorTerm = rating -
DotProduct(&h_movieFactors[movieIndex * FACTORS], &h_userFactors[userIndex *
FACTORS], FACTORS);

                                for (int f = 0; f < FACTORS; ++f)
                                {
                                        h_movieFactors[movieIndex * FACTORS + f] +=
LEARNING_RATE * (errorTerm * h_userFactors[userIndex * FACTORS + f]
                                                - REGULARIZATION_PARAMETER *
h_movieFactors[movieIndex * FACTORS + f]);
                                }

                                for (int f = 0; f < FACTORS; ++f)
                                {
                                        h_userFactors[userIndex * FACTORS + f] +=
LEARNING_RATE * (errorTerm * h_movieFactors[movieIndex * FACTORS + f]
                                                - REGULARIZATION_PARAMETER *
h_userFactors[userIndex * FACTORS + f]);
                                }
                        }
                }

                for (int indexKnownData = 0; indexKnownData < LEARNING_DATA_SIZE;
++indexKnownData)
                {
                        int userIndex = h_learningData[indexKnownData].userIndex;
                        int movieIndex = h_learningData[indexKnownData].movieIndex;
                        int rating = h_learningData[indexKnownData].rating;
                        double errorTerm2 = rating - DotProduct(&h_movieFactors[movieIndex *
FACTORS], &h_userFactors[userIndex * FACTORS], FACTORS);
                        *h_sumErrorSquared += errorTerm2 * errorTerm2;
                }

                printf("\th_sumErrorSquared = %lf, (1/N(h_sumErrorSquared))^(1/2) = %lf\n",
*h_sumErrorSquared, sqrt((1.0/LEARNING_DATA_SIZE**h_sumErrorSquared)));
                ++iterations;
```

```
        } while (iterations < 10);//(*h_sumErrorSquared > LEARNING_DATA_SIZE *
(ERROR_THRESHOLD * ERROR_THRESHOLD));
        double endTime = get_walltime();

        printf("Performed %u iterations\n", iterations);

        printf("Total time = %lf\n", endTime - startTime);

        PrintPredictedData();

        free(h_sumErrorSquared);
        free(h_userFactors);
        free(h_movieFactors);
        free(h_learningData);

        return 0;
}
```