```
#include <sys/types.h>
#include <string>
#include <unistd.h>
#include <omp.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <sys/time.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>
#include <sys/types.h>
#include <sys/wait.h>
#include "tuples.h"
#include <vector>

#define K2_NUMCOEFFS 195582816
#define VALIDATE 0

using namespace std;

#define ABS_K3_IMP_THRESHOLD 0.5
#define ABS_K4_IMP_THRESHOLD 0.35




//constant memory
float* __constant__ geneData_dev[2];
float* __constant__ tfData_dev[2];
/*float* __constant__ results000_dev[2];
float* __constant__ results002_dev[2];
float* __constant__ results022_dev[2];
float* __constant__ results222_dev[2];
float* __constant__ resultstf4_dev[2];
float* __constant__ resultscomp_dev[2];*/



//float*  geneData_dev[2];
//float*  tfData_dev[2];

/*float* __constant__ geneData_dev1=NULL;
float* __constant__ tfData_dev1=NULL;*/


/*
CPU Routines
*/
bool allFalse(bool*,int);
void dumpData(int,int,float*,char**);
int readDataFromSTDIN(int,int,float*,char**);
inline float floatCorrel(float*,float*,int);
void readRowsAndCols(int*,int*,int*);
void expTest(void);
void mulAlpha(float,float*,int);
float singleCUDACorrelWrapper(float*,float*,int);
float* k1Analysis(float*,char**,float*,char**,int,int,int,int,int,int);
float* k2Analysis(float*,char**,float*,char**,int,int,int,int,int,int);
inline double getDoubleTimeDiff(timeval t1,timeval t2);
inline double getDoubleTime(timeval t);
inline double computeSpeedup(double ,double newTime);
inline float floatKCorrel(float* ,float* ,int ,int ,int ,int ,int ,int ,int );
inline float floatKCorrelHOB(float* ,float* ,int ,int ,int ,int ,int ,int ,int ,int );
inline float accessCoeff(float* ,int ,int ,int );
inline void condFree(void*);
inline void condCUDAFree(void*);
float* k2AnalysisMultiGPU(float*,char**,float*,char**,int,int,int,int,int,int,int,dim3,dim3);
```

```
float* k2AnalysisMultiGPUBLOCK(float*,char**,float*,char**,int,int,int,int,int,int);
bool copyK2JobsToDev(int,int,int,int,int,int*,int*,int*,int,cudaStream_t);
float* k3Analysis(float*,char**,float*,char**,int,int,int,int,int,int,int);
void cFoscJunValidationAnalysis(float*,char**,float*,char**,int ,int ,int,bool );
void k3AnalysisMultiGPU(float*,char**,float*,char**,int,int,int,int,int,int);
void k4AnalysisMultiGPU(float*,char**,float*,char**,int,int,int,int,int,int);
inline bool cycleInt(int& n,int c);
inline int returnk2Index(int,int);
inline void k3Function(int,float*,float*,int,int,int,int,int,int,int,float*,int*);
inline float calcK3AbsImp(float,float,float,float,float,float,float);
inline int returnk3Index(int,int,int);
void k4Analysis(float*,char**,float*,char**,int,int,int,int,int,int,int);
void k4AnalysiNOComparisonForAbsImp(float*,char**,float*,char**,int,int,int,int,int,int,int);
inline float calcK4AbsImp(float,float,float,float,float,float,float,float,float,float,float,float,float,float,float,float);
inline bool pairNeedsCPUComp(int,int);
inline bool tripletNeedsCPUComp(int,int,int);
inline k3TFTriple returnTupleOffset(int,int,int);
int getNextGeneDataIndex(void);
int getGeneDataIndexDelta(void);
void minik4(int g,int tf1,int tf2,int tf3,int tf4,float* geneData,float* tfData,char** geneNames,char** tfNames,int numDataCols);
bool isHOBTissueIndex(int);




/*
CUDA Routines
*/
__device__ void SingleCUDACorrelDEVICE(float* ,float* ,int ,float* );
__global__ void CUDAk11WayBatch(float* ,float* ,int ,int ,float*,bool*);
__device__ void singleCUDAk2CorrelDEVICE(float*,float*,float*,int,float*);
__global__ void GPUk2TupleFunction(int,int,int,int,float*,float*,float*,int*,int*,int*,int);
__device__ bool twoTuplesEqualGPU(struct k2Tuple,struct k2Tuple);
__device__ void inck2TupleGPU(struct k2Tuple*,int);
__global__ void GPUk2BlockTupleFunction(int,int,int,float*,float*,float*,int);
__device__ void singleCUDAk3CorrelDEVICE(float*,float*,float*,float*,int,float*);
__global__ void gpu3kFunction(int,float*,float*,int,float*,int,int,int);
__global__ void gpu4kFunction(int,float*,float*,int,float*,int,int,int,int);
__device__ void singleCUDAk4CorrelDEVICE(float*,float*,float*,float*,float*,int,float*);




int main(int numArgs,char** args)
        {
#define NUM_NAMES 5000
#define MAX_NAME_SIZE 32

         //declare variables
         int index=0;
         int numTFDataPoints=0;
         int numGeneDataPoints=0;
         float* geneData;//just expression data
         float* tfData;//just expression data
         char* tfNames[NUM_NAMES];
         char* geneNames[NUM_NAMES];
         bool memAllocationSuccessful=true;
         int numRowsTFFile=0;
         int numRowsGeneFile=0;
         int numCols=0;
         int numDataCols=0;
         timeval mytimeVals;
         timeval mytimeVale;
         double cpuTime=1.0;
         double gpuTime=1.0;
         float* k1ResultsCPU=NULL;
         float* k2ResultsCPU=NULL;
         float* k1ResultsGPU=NULL;
         float* k2ResultsGPU=NULL;
         //int gpuToWorkOn=0;
         //float* k3Grid=NULL;
```

```
int numGenes;
int numTFs;

printf("*****************************************\n");
printf("Main called numArgs=%d\n",numArgs);
printf("PID = %d\n",getpid());
for(int ta=0;ta<numArgs;ta++)
          {
          printf("Arg[%d]=%s\n",ta,args[ta]);
          }
printf("*****************************************\n");


//read row/column data
readRowsAndCols(&numRowsGeneFile,&numRowsTFFile,&numCols);
numGenes=numRowsGeneFile;
numDataCols=numCols-1;//numCols includes row header strings
printf("numDataCols is %d. (less row headers)\n",numDataCols);


//calcuate number of data points
numTFDataPoints=(numDataCols)*numRowsTFFile;
numTFs=numRowsTFFile;
numGeneDataPoints=(numDataCols)*numRowsGeneFile;
printf("%d data points needed for gene data.\n",numGeneDataPoints);
printf("%d data points needed for tf data.\n",numTFDataPoints);

//allocate space for names
for(index=0;index<NUM_NAMES;index++)
          {
          tfNames[index]=(char*)(malloc(MAX_NAME_SIZE*sizeof(char)));
          if(tfNames[index]==NULL)
                    {
                    memAllocationSuccessful=false;
                    }
          geneNames[index]=(char*)(malloc(MAX_NAME_SIZE*sizeof(char)));
          if(geneNames[index]==NULL)
                    {
                    memAllocationSuccessful=false;
                    }
          }
//allocate space for data
tfData=(float*)(malloc(numTFDataPoints*sizeof(float)));
geneData=(float*)(malloc(numGeneDataPoints*sizeof(float)));
if(tfData==NULL || geneData==NULL)
          {
          memAllocationSuccessful=false;
          }


if(memAllocationSuccessful)
          {
          //run program
          printf("Loading data for gene file ...");
          readDataFromSTDIN(numRowsGeneFile,numCols,geneData,geneNames);
          printf("done!\n");
          //dumpData(numRowsGeneFile,numCols,geneData,geneNames);
          //printf("Now transforming gene data with exp ...");
          //mulAlpha(0.5,geneData,numGeneDataPoints);
          //printf("done!\n");
          //dumpData(numRowsGeneFile,numCols,geneData,geneNames);
          printf("Loading data for tf file ...");
          readDataFromSTDIN(numRowsTFFile,numCols,tfData,tfNames);
          printf("done!\n");



          //cFoscJunValidationAnalysis(geneData,geneNames,tfData,tfNames,numGenes,numTFs,numTFDataPoints,numDataCols,true);


          gettimeofday(&mytimeVals,NULL);
          k4AnalysisMultiGPU(geneData,geneNames,tfData,tfNames,numGenes,numGeneDataPoints,numTFs,numTFDataPoints,numDataCols,2);
```

```
		for(int t=4;t>=1;t--)
			{
			printf("Performing a k=4 analysis on one gene with %d thread(s)...\n",t);
			gettimeofday(&mytimeVals,NULL);
			for(int g=0;g<1;g++)
				{
				k4Analysis(
					geneData,
					geneNames,
					tfData,
					tfNames,
					numRowsGeneFile,
					numGeneDataPoints,
					numRowsTFFile,
					numTFDataPoints,
					numDataCols,
					g,//gene num
					t//numthreads
					);
				}//for loop of gene indices
			gettimeofday(&mytimeVale,NULL);
			cpuTime=getDoubleTimeDiff(mytimeVals,mytimeVale);
			printf("time with t=%d  threads :\t%f\n",t,cpuTime);
			printf("Speedup GPU/CPU        :\t%f \n",computeSpeedup(cpuTime,gpuTime));
			}



		gettimeofday(&mytimeVale,NULL);
		gpuTime=getDoubleTimeDiff(mytimeVals,mytimeVale);
		//gpuTime=1.0;
		printf("Time for 2 genes with 2 GTX590 GPUs : %f\n",gpuTime);


		condFree(k1ResultsCPU);
		condFree(k1ResultsGPU);
		condFree(k2ResultsCPU);
		condFree(k2ResultsGPU);
		}
	else
		{
		printf("MEMORY ALLOCATION FAILURE!\n");
		}

	//deallocate memory for names
	for(index=0;index<NUM_NAMES;index++)
		{
		if(tfNames[index]!=NULL)
			{
			free(tfNames[index]);
			}
		if(geneNames[index]!=NULL)
			{
			free(geneNames[index]);
			}
		}
	//deallocate memory for data
	if(tfData!=NULL)
		{
		free(tfData);
		}
	if(geneData!=NULL)
		{
		free(geneData);
		}

	printf("Program ending normally.\n");
	return 0;
	}


/*
```

```
int getNextGeneDataIndex(void)
          {
          gdi++;
          return  gdata[gdi];
          }


int getGeneDataIndexDelta(void)
          {
          int currentIndex=gdata[gdi];
          if(currentIndex==3164)
                    {
                    return 9999;
                    }
          else
                    {
                    int nextIndex=getNextGeneDataIndex();
                    return nextIndex-currentIndex;
                    }

          }*/



bool isHOBTissueIndex(int i)
          {
          if((54<=i && i<=61) || (67<=i && i<=76))
                    {
                    //brain in 54-61
                    //heart in 67-76
                    return true;
                    }
          else
                    {
                    //not in either range
                    return false;
                    }
          }


void minik4(int geneIndex,int tf1,int tf2,int tf3,int tf4,float* geneData,float* tfData,char** geneNames,char** tfNames,int numDataCols)
          {



          float coeff1=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf1,0,0,0,0);
          float coeff2=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf2,0,0,0,0);
          float coeff12=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf2,0,0,0);
          float coeff3=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf3,0,0,0,0);
          float coeff13=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf3,0,0,0);
          float coeff23=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf2,tf3,0,0,0);
          float coeff123=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf2,tf3,0,0);
          float coeff4=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf4,0,0,0,0);
          float coeff14=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf4,0,0,0);
          float coeff24=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf2,tf4,0,0,0);
          float coeff34=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf3,tf4,0,0,0);
          float coeff124=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf2,tf4,0,0);
          float coeff234=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf2,tf3,tf4,0,0);
          float coeff134=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf3,tf4,0,0);
          float coeff1234=floatKCorrel(geneData,tfData,numDataCols,4,geneIndex,tf1,tf2,tf3,tf4,0);



          float absImp=calcK4AbsImp(
                              coeff1234,
                              coeff123,coeff124,coeff134,coeff234,
                              coeff12,coeff13,coeff14,coeff24,coeff34,coeff23,
                              coeff1,coeff2,coeff3,coeff4
                              );
```

```
          if(absImp>=ABS_K4_IMP_THRESHOLD)
                    {


                              //headers
                              printf("G\tT1\tT2\tT3\tT4\t");
                              printf("C1234\tC123\tC124\tC134\tC234\t");
                              printf("C12\tC13\tC14\tC24\tC34\tC23\t");
                              printf("C1\tC2\tC3\tC4\tAI\t");
                              printf("NG\tNT1\tNT2\tNT3\tNT4\n");
                              //values
                              printf("%d\t%d\t%d\t%d\t%d\t",geneIndex,tf1,tf2,tf3,tf4);
                              printf("%f\t%f\t%f\t%f\t%f\t",coeff1234,coeff123,coeff124,coeff134,coeff234);
                              printf("%f\t%f\t%f\t%f\t%f\t%f\t",coeff12,coeff13,coeff14,coeff24,coeff34,coeff23);
                              printf("%f\t%f\t%f\t%f\t%f\t",coeff1,coeff2,coeff3,coeff4,absImp);
                              printf("%s\t%s\t%s\t%s\t%s\n",geneNames[geneIndex],tfNames[tf1],tfNames[tf2],tfNames[tf3],tfNames[tf4]);

                    }




          }


inline k3TFTriple returnTupleOffset(int tf1,int tf2 ,int tf3)
          {

          struct k3TFTriple offsets;
          offsets.tf1=0;
          offsets.tf2=0;
          offsets.tf3=0;

          for(int a=0;a<3;a+=2)
                    {
                    for(int b=0;b<3;b+=2)
                              {
                              for(int c=0;c<3;c+=2)
                                        {
                                        if(tf1-a<350 && tf2-b<350 && tf3-c<350)
                                                  {
                                                  offsets.tf1=a;
                                                  offsets.tf2=b;
                                                  offsets.tf3=c;
                                                  return offsets;
                                                  }
                                        }
                              }
                    }

          return offsets;
          }


inline bool tripletNeedsCPUComp(int tf1,int tf2,int tf3)
          {
          if(tf1<350 && tf2<350 && tf3<350)
                    {
                    return false;
                    }
          else
                    {
                    if(tf1>=2 && tf2>=2 && tf3>=2)
                              {
                              return false;
                              }
                    else
                              {
                              return true;
                              }
                    }
          }
```

```
inline bool pairNeedsCPUComp(int tf1,int tf2)
          {

          if(tf1<2 && tf2>=350)
                    {
                    return true;
                    }
          else
                    {
                    return false;
                    }
          }



void k4Analysis(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int geneIndex,int numOpenMPThreads)
          {

          float* k3Grid=k3Analysis(geneData,geneNames,tfData,tfNames,numGenes,numGeneDataPoints,numTFs,numTFDataPoints,numDataCols,geneIndex,numOpenMPThreads);
          if(k3Grid==NULL)
                    {
                    printf("FATAL ERROR : for gene=%d, k3grid null! Bailing!\n",geneIndex);
                    return;
                    }
          else
                    {
                    printf("Successfully carried out a k3analysis as part of k=4...now resuming...\n");
                    }



          #pragma omp parallel num_threads(numOpenMPThreads) default(shared)
                    {
                    unsigned int thread_num=omp_get_thread_num();
                    unsigned int num_threads=omp_get_num_threads();
                    unsigned int x=0;
                    float coeff123,coeff124,coeff234,coeff134;
                    float coeff12,coeff13,coeff14,coeff23,coeff24,coeff34;
                    float coeff1,coeff2,coeff3,coeff4;
                    float coeff1234;
                    float absImp;
                    for(int tf1=0;tf1<350-3;tf1++)
                              {
                              printf("tf1=%d...\n",tf1);
                              for(int tf2=tf1+1;tf2<350-2;tf2++)
                                        {
                                        //printf("tf2=%d...\n",tf2);
                                        for(int tf3=tf2+1;tf3<350-1;tf3++)
                                                  {
                                                  //if(tf3%50==0 && thread_num==0) {printf("tf3=%d...\n",tf3);}
                                                  for(int tf4=tf3+1;tf4<350;tf4++)
                                                            {
                                                            if(x%num_threads==thread_num)
                                                                      {
                                                                      //k=4
                                                                      coeff1234=floatKCorrel(geneData,tfData,numDataCols,4,geneIndex,tf1,tf2,tf3,tf4,0);

                                                                      //k=3
                                                                      coeff124=k3Grid[returnk3Index(tf1,tf2,tf4)];
                                                                      coeff234=k3Grid[returnk3Index(tf2,tf3,tf4)];
                                                                      coeff134=k3Grid[returnk3Index(tf1,tf3,tf4)];
                                                                      coeff123=k3Grid[returnk3Index(tf1,tf2,tf3)];

                                                                      //k=2
                                                                      coeff14=k3Grid[returnk2Index(tf1,tf4)];
                                                                      coeff24=k3Grid[returnk2Index(tf2,tf4)];
                                                                      coeff34=k3Grid[returnk2Index(tf3,tf4)];
                                                                      coeff13=k3Grid[returnk2Index(tf1,tf3)];
                                                                      coeff12=k3Grid[returnk2Index(tf1,tf2)];
                                                                      coeff23=k3Grid[returnk2Index(tf2,tf3)];
```

```
                                                  //k=1
                                                  coeff1=k3Grid[returnk2Index(tf1,tf1)];
                                                  coeff2=k3Grid[returnk2Index(tf2,tf2)];
                                                  coeff3=k3Grid[returnk2Index(tf3,tf3)];
                                                  coeff4=k3Grid[returnk2Index(tf4,tf4)];

                                                  absImp=calcK4AbsImp(
                                                                    coeff1234,
                                                                    coeff123,coeff124,coeff134,coeff234,
                                                                    coeff12,coeff13,coeff14,coeff24,coeff34,coeff23,
                                                                    coeff1,coeff2,coeff3,coeff4
                                                                    );

                                                  if(absImp>=ABS_K4_IMP_THRESHOLD)
                                                            {
                                                            //headers
                                                            printf("G\tT1\tT2\tT3\tT4\t");
                                                            printf("C1234\tC123\tC124\tC134\tC234\t");
                                                            printf("C12\tC13\tC14\tC24\tC34\tC23\t");
                                                            printf("C1\tC2\tC3\tC4\tAI\t");
                                                            printf("NG\tNT1\tNT2\tNT3\tNT4\n");
                                                            //values
                                                            printf("%d\t%d\t%d\t%d\t%d\t",geneIndex,tf1,tf2,tf3,tf4);
                                                            printf("%f\t%f\t%f\t%f\t%f\t",coeff1234,coeff123,coeff124,coeff134,coeff234);
                                                            printf("%f\t%f\t%f\t%f\t%f\t",coeff12,coeff13,coeff14,coeff24,coeff34,coeff23);
                                                            printf("%f\t%f\t%f\t%f\t%f\t",coeff1,coeff2,coeff3,coeff4,absImp);
                                                            printf("%s\t%s\t%s\t%s\t%s\n",geneNames[geneIndex],tfNames[tf1],tfNames[tf2],tfNames[tf3],tfNames[tf4]);
                                                            }


                                                  }
                                        x++;
                                        }
                              }
                    }
          }
          }//openMP for 0.....350
printf("Freeing k3Grid...\n");
free(k3Grid);

//now, do just 350, 351, 352
#pragma omp parallel num_threads(numOpenMPThreads) default(shared)
          {
          unsigned int thread_num=omp_get_thread_num();
          unsigned int num_threads=omp_get_num_threads();
          unsigned int x=0;
          float coeff123,coeff124,coeff234,coeff134;
          float coeff12,coeff13,coeff14,coeff23,coeff24,coeff34;
          float coeff1,coeff2,coeff3,coeff4;
          float coeff1234;
          float absImp;
          for(int tf1=0;tf1<numTFs-3;tf1++)
                    {
                    printf("tf1=%d...\n",tf1);
                    for(int tf2=tf1+1;tf2<numTFs-2;tf2++)
                              {
                              //printf("tf2=%d...\n",tf2);
                              for(int tf3=tf2+1;tf3<numTFs-1;tf3++)
                                        {
                                        //if(tf3%50==0 && thread_num==0) {printf("tf3=%d...\n",tf3);}
                                        for(int tf4=tf3+1;tf4<numTFs;tf4++)
                                                  {
                                                  if(x%num_threads==thread_num && (tf1>=350 || tf2>=350 || tf3>=350 || tf4>=350))
                                                            {
                                                            //k=4
                                                            coeff1234=floatKCorrel(geneData,tfData,numDataCols,4,geneIndex,tf1,tf2,tf3,tf4,0);

                                                            //k=4(3s)
                                                            coeff124=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf2,tf4,0,0);
                                                            coeff234=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf2,tf3,tf4,0,0);
                                                            coeff134=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf3,tf4,0,0);
                                                            coeff123=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf2,tf3,0,0);
```

```
                                                                        //k=2
                                                                        coeff14=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf4,0,0,0);
                                                                        coeff24=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf2,tf4,0,0,0);
                                                                        coeff34=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf3,tf4,0,0,0);
                                                                        coeff13=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf3,0,0,0);
                                                                        coeff12=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf2,0,0,0);
                                                                        coeff23=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf2,tf3,0,0,0);

                                                                        //k=1
                                                                        coeff1=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf1,0,0,0,0);
                                                                        coeff2=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf2,0,0,0,0);
                                                                        coeff3=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf3,0,0,0,0);
                                                                        coeff4=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf4,0,0,0,0);


                                                                        absImp=calcK4AbsImp(
                                                                                        coeff1234,
                                                                                        coeff123,coeff124,coeff134,coeff234,
                                                                                        coeff12,coeff13,coeff14,coeff24,coeff34,coeff23,
                                                                                        coeff1,coeff2,coeff3,coeff4
                                                                                        );

                                                                        if(absImp>=ABS_K4_IMP_THRESHOLD)
                                                                                  {
                                                                                  //headers
                                                                                  printf("G\tT1\tT2\tT3\tT4\t");
                                                                                  printf("C1234\tC123\tC124\tC134\tC234\t");
                                                                                  printf("C12\tC13\tC14\tC24\tC34\tC23\t");
                                                                                  printf("C1\tC2\tC3\tC4\tAI\t");
                                                                                  printf("NG\tNT1\tNT2\tNT3\tNT4\n");
                                                                                  //values
                                                                                  printf("%d\t%d\t%d\t%d\t%d\t",geneIndex,tf1,tf2,tf3,tf4);
                                                                                  printf("%f\t%f\t%f\t%f\t%f\t",coeff1234,coeff123,coeff124,coeff134,coeff234);
                                                                                  printf("%f\t%f\t%f\t%f\t%f\t",coeff12,coeff13,coeff14,coeff24,coeff34,coeff23);
                                                                                  printf("%f\t%f\t%f\t%f\t%f\t",coeff1,coeff2,coeff3,coeff4,absImp);
                                                                                  printf("%s\t%s\t%s\t%s\t%s\n",geneNames[geneIndex],tfNames[tf1],tfNames[tf2],tfNames[tf3],tfNames[tf4]);
                                                                                  }


                                                                        }//part of this thread's job
                                                                x++;
                                                                }
                                                        }
                                                }
                                        }
                                }//openMP for 0.....350



        }




inline float calcK4AbsImp(
                    float c1234,
                    float c123,float c124,float c134,float c234,
                    float c12,float c13,float c14,float c24,float c34,float c23,
                    float c1,float c2,float c3,float c4
                    )
        {

        float absImp=
                    min(abs(c1234-c123),
                    min(abs(c1234-c124),
                    min(abs(c1234-c134),
                    min(abs(c1234-c234),
                    min(abs(c1234-c12),
                    min(abs(c1234-c13),
                    min(abs(c1234-c14),
                    min(abs(c1234-c24),
                    min(abs(c1234-c34),
```

```
                    min(abs(c1234-c23),
                    min(abs(c1234-c1),
                    min(abs(c1234-c2),
                    min(abs(c1234-c3),
                    abs(c1234-c4)))))))))))));

          return absImp;
          }


inline int returnk3Index(int tf1,int tf2,int tf3)
          {

          //find the coordinates
          int gridx=floor(tf1/10);
          int gridy=floor(tf2/10);
          int gridz=floor(tf3/10);
          int blockx=tf1%10;
          int blocky=tf2%10;
          int blockz=tf3%10;

          //compute threadID in a block index (idx), blockId in a grid (bdx), and threadID overall (idx)
          int idx=blockx+blocky*10+blockz*10*10;                                    //threadID in a block (0-999)
          int bdx=gridx +gridy *35+gridz*35 *35;  //blockID in a grid;  (0-35^3)
          idx=(10*10*10)*bdx+idx;

          return idx;
          }


inline int returnk2Index(int tf1,int tf2)
          {
          /*int tf1=min(tfa,tfb);
          int tf2=max(tfa,tfb);*/


          int gridz=floor(tf2/10);
          int gridy=floor(tf1/10);
          int gridx=floor(tf1/10);
          int blockx=tf1%10;
          int blocky=tf1%10;
          int blockz=tf2%10;



          int index=gridz*1225000;//1000*35^2
          index=index+35000*gridy;//1000*35
          index=index+1000*gridx; //1000
          index=index+100*blockz;
          index=index+10*blocky;
          index=index+blockx;



          //if(tf1==100 && tf2==100)
          if(0==1)
                    {
                    printf("For a point tf1=%d, tf2=%d, \n",tf1,tf2);
                    printf("\tgx=%d\tgy=%d\tgz=%d\tbx=%d\tby=%d\tbz=%d\n",
                    gridx,gridy,gridz,blockx,blocky,blockz);
                    printf("The index to-be-returned : %d\n",index);
                    printf("\n\n\n");
                    }

          return index;
          }



inline bool cycleInt(int& n,int c)
          {
          n++;
```

```
          if(n>=c)
                    {
                    n=0;
                    return true;
                    }
          else
                    {
                    return false;
                    }
          }



void condFree(void* p)
          {
          if(p!=NULL)
                    {
                    free(p);
                    }
          }


inline float accessCoeff(float* data,int g,int t,int numTFs)
          {
          //int index=g*numTFs+t;
          return data[g*numTFs+t];
          }



inline double computeSpeedup(double oldTime,double newTime)
          {
          double s=(oldTime/newTime);
          return s;
          }


inline double getDoubleTimeDiff(timeval t1,timeval t2)
          {
          double t1d=getDoubleTime(t1);
          double t2d=getDoubleTime(t2);
          double diff=t1d-t2d;
          if(diff<0)
                    {
                    diff=(-1)*diff;
                    }
          return diff;
          }

inline double getDoubleTime(timeval t)
          {
          long s=t.tv_sec;
          long ms=t.tv_usec;
          double dms=(double)((double)(ms)/(double)(1000000.0));
          double all=(double)(s)+dms;
          return all;
          }

void k4AnalysisMultiGPU(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int numGPUs)
          {

          /*
          alternate GPUs with each call to a kernel being a full k=3 analysis, but with
          a different gene.  use 3-dimensional blocks 10x10x10=1000 threads/block
          Use 3-D grids as well...that may speed things up too..
          */


          int gpu=0;
          float* resultData000[2];
          float* resultData222[2];
          float* resultData022[2];
```

```
	float* resultData002[2];
	float* results000=NULL;
	float* results002=NULL;
	float* results022=NULL;
	float* results222=NULL;
	float* resultData_dev[2];
	cudaStream_t cudaStreams[2];
	int  numCoeffsPerKernel=350*350*350;
	bool allocError=false;


	//INITIALIZE!!!!
	for(gpu=0;gpu<numGPUs;gpu++)
	//for(gpu=gpuStartOn;gpu<numGPUs; gpu=gpu+(stayOnGPU?1:0))
		{
		//NULLify
		geneData_dev[gpu]=NULL;
		tfData_dev[gpu]=NULL;
		resultData000[gpu]=NULL;
		resultData002[gpu]=NULL;
		resultData022[gpu]=NULL;
		resultData222[gpu]=NULL;
		resultData_dev[gpu]=NULL;
		cudaStreams[gpu]=NULL;
		/*results000_dev[gpu]=NULL;
		results002_dev[gpu]=NULL;
		results022_dev[gpu]=NULL;
		results222_dev[gpu]=NULL;
		resultscomp_dev[gpu]=NULL;
		resultstf4_dev[gpu]=NULL;*/

		//set the proper device
		if(cudaSuccess==cudaSetDevice(gpu))
			{
			//STREAMS
			//printf("gstream...\n");
			if(cudaSuccess!=cudaStreamCreate(&(cudaStreams[gpu])))
				{
				printf("Error allocating a stream for gpu=%d!\n",gpu);
				allocError=true;
				}
			else
				{
				printf("Stream #%d : %p \n",gpu,cudaStreams[gpu]);
				}

			//GENE DATA
			//printf("cgd...\n");
			if(cudaSuccess!=cudaMalloc((void**)(&(geneData_dev[gpu])),numGeneDataPoints*sizeof(float)))
				{
				printf("GPU %d had error in allocating device memory for gene data!\n",gpu);
				geneData_dev[gpu]=NULL;
				allocError=true;
				}
			else
				{
				//copy gene data to device
				if(cudaSuccess!=cudaMemcpyAsync(geneData_dev[gpu],
								geneData,numGeneDataPoints*sizeof(float),
								cudaMemcpyHostToDevice,cudaStreams[gpu]))
					{
					printf("ERROR (g=%d) in copying gene data to device!\n",gpu);
					allocError=true;
					}
				}

			//TF DATA
			//printf("tfd...\n");
			if(cudaSuccess!=cudaMalloc((void**)(&(tfData_dev[gpu])),numTFDataPoints*sizeof(float)))
				{
```

```
                printf("ERROR (g=%d) allocating CUDA memory for TF data!\n",gpu);
                tfData_dev[gpu]=NULL;
                allocError=true;
                }
        else
                {
                //copy TF data to device
                if(cudaSuccess!=cudaMemcpyAsync(tfData_dev[gpu],
                                                tfData,numTFDataPoints*sizeof(float),
                                                cudaMemcpyHostToDevice,
                                                cudaStreams[gpu]))
                        {
                        printf("ERROR (g=%d) copying TF data to device memory!\n",gpu);
                        allocError=true;
                        }
                }


        //HOST data
        //printf("hd...\n");
        //000
        if(cudaSuccess!=cudaMallocHost((void**)&(resultData000[gpu]),numCoeffsPerKernel*sizeof(float)))
                {
                printf("ERROR (g=%d) in allocating page-locked memory!!!\n",gpu);
                resultData000[gpu]=NULL;
                allocError=true;
                }
        else
                {
                //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                }
        //222
        if(cudaSuccess!=cudaMallocHost((void**)&(resultData222[gpu]),numCoeffsPerKernel*sizeof(float)))
                {
                printf("ERROR (g=%d) in allocating page-locked memory P2!!!\n",gpu);
                resultData222[gpu]=NULL;
                allocError=true;
                }
        else
                {
                //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                }
        //002
        if(cudaSuccess!=cudaMallocHost((void**)&(resultData002[gpu]),numCoeffsPerKernel*sizeof(float)))
                {
                printf("ERROR (g=%d) in allocating page-locked memory P2!!!\n",gpu);
                resultData002[gpu]=NULL;
                allocError=true;
                }
        else
                {
                //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                }
        //022
        if(cudaSuccess!=cudaMallocHost((void**)&(resultData022[gpu]),numCoeffsPerKernel*sizeof(float)))
                {
                printf("ERROR (g=%d) in allocating page-locked memory P2!!!\n",gpu);
                resultData022[gpu]=NULL;
                allocError=true;
                }
        else
                {
                //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                }

        //GPU/device data
        //printf("devd...\n");
```

```
                        if(cudaSuccess!=cudaMalloc((void**)(&(resultData_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                allocError=true;
                                resultData_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results!\n",gpu);
                                }

                        /*000 dev data
                        if(cudaSuccess!=cudaMalloc((void**)(&(results000_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                allocError=true;
                                results000_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results 000!\n",gpu);
                                }

                        //002 dev data
                        if(cudaSuccess!=cudaMalloc((void**)(&(results002_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                allocError=true;
                                results002_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results 002!\n",gpu);
                                }


                        //022 dev data
                        if(cudaSuccess!=cudaMalloc((void**)(&(results022_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                allocError=true;
                                results022_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results 022!\n",gpu);
                                }

                        //222 dev data
                        if(cudaSuccess!=cudaMalloc((void**)(&(results222_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                allocError=true;
                                results222_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results 222!\n",gpu);
                                }

                        if(cudaSuccess!=cudaMalloc((void**)(&(resultstf4_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                resultstf4_dev[gpu]=NULL;
                                allocError=true;
                                printf("GPU %d had error in allocating space for TF4 results!\n",gpu);
                                }


                        //comp results dev
                        int size=621682425;
                        while(resultscomp_dev[gpu]==NULL)
                                {
                                if(cudaSuccess!=cudaMalloc((void**)(&(resultscomp_dev[gpu])),size*sizeof(char)))
                                        {
                                        allocError=true;
                                        resultscomp_dev[gpu]=NULL;
                                        printf("GPU %d had error in allocating device memory for comp results! size=%d\n",gpu,size);
                                        size-=100000;
                                        }
                                else
                                        {
                                        printf("GPU %d had SUCCESS in allocating device memory for comp results  size=%d\n",gpu,size);
                                        }
                                }*/



                        }//set cuda device!
                else
                        {
```

```
                        allocError=true;
                        printf("Error in selecting CUDA device %d!\n",gpu);
                        }
                }  //allocate memories, per host, per GPU (data, results, streams)


        if(!allocError)
                {

                //10x10x10 blocks
                //35x35x35 grid!
                //Maximum sizes of each dimension of a block:    1024 x 1024 x 64
                //Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
                dim3 dimBlock(10,10,10);
                dim3 dimGrid(35,35,35);
                int geneIndex=0;
                for(int g=0;g<numGenes;g+=numGPUs)
                        {
                        printf("Doing a process loop g=%d,%d...\n",g,g+numGPUs-1);

                        for(int gpu=0;gpu<numGPUs;gpu++)
                        //for(gpu=gpuStartOn;gpu<numGPUs; gpu=gpu+(stayOnGPU?1:0))
                                {
                                if(gpu<numGenes)
                                        {
                                        geneIndex=gpu+g;
                                        printf("Now process gene=%d with gpu=%d!\n",gpu+g,gpu);
                                        if(cudaSetDevice(gpu)==cudaSuccess)
                                                {
                                                ////////////////////////////////////////call kernel (stream) for gene gpu+g (geneIndex)  (000)
                                                gpu3kFunction<<<
                                                        dimGrid,dimBlock,0,cudaStreams[gpu]
                                                        >>>(
                                                        g+gpu,
                                                        geneData_dev[gpu],
                                                        tfData_dev[gpu],
                                                        numDataCols,
                                                        resultData_dev[gpu],
                                                        0,0,0
                                                        );
                                                //copy back results (stream)
                                                if(cudaSuccess!=cudaMemcpyAsync(
                                                        resultData000[gpu],
                                                        resultData_dev[gpu],
                                                        numCoeffsPerKernel*sizeof(float),
                                                        cudaMemcpyDeviceToHost,
                                                        cudaStreams[gpu]))
                                                        {
                                                        printf("Error (000) in copying back data from k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                        }


                                                ////////////////////////////////////////////call kernel (stream) for gene gpu+g (geneIndex)  (002)
                                                gpu3kFunction<<<
                                                        dimGrid,dimBlock,0,cudaStreams[gpu]
                                                        >>>(
                                                        g+gpu,
                                                        geneData_dev[gpu],
                                                        tfData_dev[gpu],
                                                        numDataCols,
                                                        resultData_dev[gpu],
                                                        0,0,2//0->2,...,349->351
                                                        );
                                                //copy back results (stream)
                                                if(cudaSuccess!=cudaMemcpyAsync(
                                                        resultData002[gpu],
                                                        resultData_dev[gpu],
                                                        numCoeffsPerKernel*sizeof(float),
                                                        cudaMemcpyDeviceToHost,
```

```
                                                cudaStreams[gpu]))
                                                {
                                                printf("Error (002) in copying back data from k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                }


                                                ////////////////////////////////////////////call kernel (stream) for gene gpu+g (geneIndex)  (022)
                                                gpu3kFunction<<<
                                                          dimGrid,dimBlock,0,cudaStreams[gpu]
                                                          >>>(
                                                          g+gpu,
                                                          geneData_dev[gpu],
                                                          tfData_dev[gpu],
                                                          numDataCols,
                                                          resultData_dev[gpu],
                                                          0,2,2//0->2,...,349->351
                                                          );
                                                //copy back results (stream)
                                                if(cudaSuccess!=cudaMemcpyAsync(
                                                          resultData022[gpu],
                                                          resultData_dev[gpu],
                                                          numCoeffsPerKernel*sizeof(float),
                                                          cudaMemcpyDeviceToHost,
                                                          cudaStreams[gpu]))
                                                          {
                                                          printf("Error  (022) in copying back data from k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                          }


                                                ////////////////////////////////////////////call kernel (stream) for gene gpu+g (geneIndex)  (222)
                                                gpu3kFunction<<<
                                                          dimGrid,dimBlock,0,cudaStreams[gpu]
                                                          >>>(
                                                          g+gpu,
                                                          geneData_dev[gpu],
                                                          tfData_dev[gpu],
                                                          numDataCols,
                                                          resultData_dev[gpu],
                                                          2,2,2//0->2,...,349->351
                                                          );
                                                //copy back results (stream)
                                                if(cudaSuccess!=cudaMemcpyAsync(
                                                          resultData222[gpu],
                                                          resultData_dev[gpu],
                                                          numCoeffsPerKernel*sizeof(float),
                                                          cudaMemcpyDeviceToHost,
                                                          cudaStreams[gpu]))
                                                          {
                                                          printf("Error  (222) in copying back data from k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                          }


                                                }
                                          else
                                                {
                                                printf("ERROR in selecting a CUDA device for k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                }
                                     }  //gpu<numGenes (if there are more GPUs than GENEs this block prevents a GPU from trying to access a gene that doesn't exist for it
                               }  // for gpu<numGPUs for loop


                    for(int gpu=0;gpu<numGPUs;gpu++)
                    //for(gpu=gpuStartOn;gpu<numGPUs; gpu=gpu+(stayOnGPU?1:0))
                              {
                              //synchronize on stream
```

```
                                        //process results for gene g from gpu!

                                        if(cudaSetDevice(gpu)==cudaSuccess)
                                                {
                                                printf("Success in setting gpu=%d...\n",gpu);
                                                if(cudaStreamSynchronize(cudaStreams[gpu])==cudaSuccess)
                                                        {
                                                        results000=resultData000[gpu];
                                                        results022=resultData022[gpu];
                                                        results002=resultData002[gpu];
                                                        results222=resultData222[gpu];
                                                        float* gpuTF4m[4];

                                                        for(int talloc=0;talloc<4;talloc++)
                                                                {
                                                                gpuTF4m[talloc]=NULL;
                                                                //printf("Before gputf4m[t]=gputf4m[%d]=%p...\n",talloc,gpuTF4m[talloc]);
                                                                if(cudaSuccess!=cudaMallocHost((void**)&(gpuTF4m[talloc]),numCoeffsPerKernel*sizeof(float)))
                                                                        {
                                                                        printf("ERROR (g=%d,thread=%d) in allocating page-locked memory special TF4!!!\n",gpu,talloc);
                                                                        }
                                                                else
                                                                        {
                                                                        //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",talloc);
                                                                        //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                                                                        }
                                                                //printf("After gputf4m[t]=gputf4m[%d]=%p...\n",talloc,gpuTF4m[talloc]);
                                                                }

                                                        #pragma omp parallel num_threads(4) default(none)
shared(gpuTF4m,numTFs,cudaStreams,resultData_dev,dimBlock,dimGrid,tfData_dev,geneData_dev,numCoeffsPerKernel,results000,results002,results222,results022,g,gpu,numDataCols,tfData,geneData,t
fNames,geneNames)
                                                        {
                                                        //int x=0;
                                                        int thread_num=omp_get_thread_num();
                                                        int num_threads=omp_get_num_threads();
                                                        float coeff1234;
                                                        float coeff123,coeff124,coeff134,coeff234;
                                                        float coeff12,coeff13,coeff14,coeff23,coeff24,coeff34;
                                                        float coeff1,coeff2,coeff3,coeff4;
                                                        //int startTF2=1;

                                                        float absImp;
                                                        float* gpuTF4=gpuTF4m[thread_num];

                                                        for(int tf1=thread_num;tf1<numTFs-3;tf1+=num_threads)
                                                                {
                                                                //startTF2=tf1+1;
                                                                //tf1 in 0....348
                                                                //if((x%num_threads)==thread_num)       {//c1
                                                                        coeff1=results000[returnk2Index(tf1,tf1)];

                                                                        #pragma omp critical
                                                                                {
                                                                                printf("Thread %d (of %d) working on set with TF1=%d....\n",thread_num,num_threads,tf1);

                                                                                //printf("\n\n\n");
                                                                                //call the special kernel
                                                                                #define K4BOUNDARY 255

                                                                                if(tf1<K4BOUNDARY)
                                                                                        {
                                                                                        if(cudaSetDevice(gpu)!=cudaSuccess)
                                                                                                {
                                                                                                printf("Error in selecting a GPU device in an openMP thread!\n");
                                                                                                }

                                                                                        gpu4kFunction<<<dimGrid,dimBlock,0,cudaStreams[gpu]>>>(
                                                                                                g+gpu,
                                                                                                geneData_dev[gpu],
```

```
                                                                                tfData_dev[gpu],
                                                                                numDataCols,
                                                                                resultData_dev[gpu],
                                                                                1,2,2,
                                                                                tf1
                                                                                );

                                                                        //copy back results (stream)
                                                                        if(cudaSuccess!=cudaMemcpyAsync(
                                                                                        gpuTF4,
                                                                                        resultData_dev[gpu],
                                                                                        numCoeffsPerKernel*sizeof(float),
                                                                                        cudaMemcpyDeviceToHost,
                                                                                        cudaStreams[gpu]
                                                                                        ))
                                                                                {
                                                                                printf("Thread %d had error (gpu) in copying back data for a k=4
analysis! g=%d, gpu=%d, tf1=%d!\n",thread_num,g+gpu,gpu,tf1);
                                                                                printf("gputf4 : %p\n",gpuTF4);
                                                                                printf("stream : %p\n",cudaStreams[gpu]);
                                                                                if(cudaSuccess != cudaGetLastError())
                                                                                        printf("Errors String : '%s'\n",
cudaGetErrorString(cudaGetLastError()));
                                                                                tf1=numTFs;
                                                                                }

                                                                        if(cudaStreamSynchronize(cudaStreams[gpu])!=cudaSuccess)
                                                                                {
                                                                                printf("FATAL ERROR IN SYNCH!  in copying back data for a k=4
analysis! g=%d, gpu=%d, tf1=%d!\n",g+gpu,gpu,tf1);
                                                                                tf1=numTFs;
                                                                                }


                                                                        ///printf("According to GPU coeff 1234 is %f\n",gpuTF4[returnk3Index(340-1,341-
1,342-1)]);
                                                                        //printf("CPU is
%f\n",floatKCorrel(geneData,tfData,numDataCols,4,g+gpu,tf1,340,341,342,0));
                                                                        //printf("\n\n\n");
                                                                        }//GPU USED WHEN TF1<BOUNDARY  CRITICAL SECTION
                                                                }
                                                        //}//realm of work for this thread

                                                for(int tf2=tf1+1;tf2<numTFs-2;tf2++)
                                                        {//tf2 in 1.....349

                                                        //if((x%num_threads)==thread_num) {//c2,c12

                                                                coeff2=results000[returnk2Index(tf2,tf2)];
                                                                coeff12=results000[returnk2Index(tf1,tf2)];
                                                                //}//in this threads's realm of work

                                                        for(int tf3=tf2+1;tf3<numTFs-1;tf3++)
                                                                {//tf3 in 2...350

                                                                //if((x%num_threads)==thread_num) {//c3,c13,c23,c123
                                                                        coeff3=results222[returnk3Index(tf3-2,tf3-2,tf3-2)];

                                                                        //c13
                                                                        if(tf3<350)
                                                                                {
                                                                                coeff13=results000[returnk2Index(tf1,tf3)];
                                                                                }
                                                                        else
                                                                                {
                                                                                coeff13=results022[returnk3Index(tf1-0,tf3-2,tf3-2)];
                                                                                }

                                                                        //c23
                                                                        if(tf3<350)
```

```
                {
                //printf("getting c23 from base 1...\n");
                coeff23=results000[returnk2Index(tf2,tf3)];
                }
        else
                {
                //printf("getting c23 from base 2...\n");
                coeff23=results002[returnk2Index(tf2-0,tf3-2)];
                }

        //c123
        if(tf3<350)
                {
                coeff123=results000[returnk3Index(tf1,tf2,tf3)];
                }
        else if(tf2<350)
                {
                coeff123=results002[returnk3Index(tf1,tf2-0,tf3-2)];
                }
        else
                {
                coeff123=results022[returnk3Index(tf1,tf2-2,tf3-2)];
                }

        //}//in the real of this thread's work

for(int tf4=tf3+1;tf4<numTFs;tf4++)
          {//tf4 in 3....351
          //if((x%num_threads)==thread_num) {

                //c4
                #pragma omp critical
                if(tf4>=350)
                        {
                        coeff4=results222[returnk3Index(tf4-2,tf4-2,tf4-2)];
                        }
                else
                        {
                        coeff4=results000[returnk3Index(tf4-0,tf4-0,tf4-0)];
                        }

                //c14
                if(tf4<350)
                        {
                        coeff14=results000[returnk2Index(tf1,tf4)];
                        }
                else
                        {
                        coeff14=results022[returnk3Index(tf1-0,tf4-2,tf4-2)];
                        }

                //c24
                if(tf4<350)
                        {
                        coeff24=results000[returnk2Index(tf2,tf4)];
                        }
                else
                        {
                        coeff24=results022[returnk3Index(tf2-0,tf4-2,tf4-2)];
                        }

                //c34
                if(tf4<350)
                        {
                        coeff34=results000[returnk2Index(tf3,tf4)];
                        }
                else
                        {
```

```
                                                                coeff34=results222[returnk3Index(tf3-2,tf4-2,tf4-2)];
                                                                }

                                                        //c124
                                                        if(tf4<350)
                                                                {
                                                                coeff124=results000[returnk3Index(tf1,tf2,tf4)];
                                                                }
                                                        else if(tf1>=2)
                                                                {
                                                                coeff124=results222[returnk3Index(tf1-2,tf2-2,tf4-2)];
                                                                }
                                                        else
                                                                {
                                                                coeff124=results002[returnk3Index(tf1-0,tf2-0,tf4-2)];
                                                                }

                                                        //c234
                                                        if(tf4<350)
                                                                {
                                                                coeff234=results000[returnk3Index(tf2,tf3,tf4)];
                                                                }
                                                        else if(tf2<2)
                                                                {
                                                                coeff234=results022[returnk3Index(tf2-0,tf3-2,tf4-2)];
                                                                }
                                                        else
                                                                {
                                                                coeff234=results222[returnk3Index(tf2-2,tf3-2,tf4-2)];
                                                                }

                                                        //c134
                                                        if(tf4<350)
                                                                {
                                                                coeff134=results000[returnk3Index(tf1,tf3,tf4)];
                                                                }
                                                        else if(tf1<2)
                                                                {
                                                                coeff134=results022[returnk3Index(tf1-0,tf3-2,tf4-2)];
                                                                }
                                                        else
                                                                {
                                                                coeff134=results222[returnk3Index(tf1-2,tf3-2,tf4-2)];
                                                                }

                                                        //c1234
                                                        if(tf1<K4BOUNDARY)
                                                                {
                                                                /*#pragma omp critical
                                                                        {
                                                                        printf("c1234 from gpu! t=%d %d %d %d
%d\n",thread_num,tf1,tf2,tf3,tf4);
                                                                        }*/
                                                                coeff1234=gpuTF4[returnk3Index(tf2-1,tf3-2,tf4-2)];
                                                                }//GPU USED FOR K=4 WHEN TF1<BOUNDARY
                                                        else
                                                                {
                                                                /*#pragma omp critical
                                                                        {
                                                                        printf("c1234 from cpu! t=%d %d %d %d
%d\n",thread_num,tf1,tf2,tf3,tf4);
                                                                        }*/

                coeff1234=floatKCorrel(geneData,tfData,numDataCols,4,g+gpu,tf1,tf2,tf3,tf4,0);
                                                                }

                                                        absImp=calcK4AbsImp(
```

```
coeff1234,
coeff123,coeff124,coeff134,coeff234,
coeff12,coeff13,coeff14,coeff24,coeff34,coeff23,
coeff1,coeff2,coeff3,coeff4
);

if(absImp>=ABS_K4_IMP_THRESHOLD)
{
#pragma omp critical
{
//headers
printf("G\tT1\tT2\tT3\tT4\t");
printf("C1234\tC123\tC124\tC134\tC234\t");
printf("C12\tC13\tC14\tC24\tC34\tC23\t");
printf("C1\tC2\tC3\tC4\tAI\t");
printf("NG\tNT1\tNT2\tNT3\tNT4\n");
//values
printf("%d\t%d\t%d\t%d\t
%d\t",g+gpu,tf1,tf2,tf3,tf4);
printf("%f\t%f\t%f\t%f\t
%f\t",coeff1234,coeff123,coeff124,coeff134,coeff234);
printf("%f\t%f\t%f\t%f\t%f\t
%f\t",coeff12,coeff13,coeff14,coeff24,coeff34,coeff23);
printf("%f\t%f\t%f\t%f\t
%f\t",coeff1,coeff2,coeff3,coeff4,absImp);
printf("%s\t%s\t%s\t%s\t
%s\n",geneNames[g+gpu],tfNames[tf1],tfNames[tf2],tfNames[tf3],tfNames[tf4]);
}
}

if(VALIDATE)
{

if(
abs(coeff1-
floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf1,0,0,0,0))>=0.01 ||
abs(coeff2-
floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf2,0,0,0,0))>=0.01 ||
abs(coeff3-
floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf3,0,0,0,0))>=0.01 ||
abs(coeff4-
floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf4,0,0,0,0))>=0.01 ||

abs(coeff12-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf2,0,0,0))>=0.01 ||
abs(coeff13-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf3,0,0,0))>=0.01 ||
abs(coeff14-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf4,0,0,0))>=0.01 ||
abs(coeff23-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf2,tf3,0,0,0))>=0.01 ||
abs(coeff24-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf2,tf4,0,0,0))>=0.01 ||
abs(coeff34-
floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf3,tf4,0,0,0))>=0.01 ||

abs(coeff123-
floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf2,tf3,0,0))>=0.01 ||
abs(coeff124-
floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf2,tf4,0,0))>=0.01 ||
abs(coeff134-
floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf3,tf4,0,0))>=0.01 ||
abs(coeff234-
floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf2,tf3,tf4,0,0))>=0.01 ||
abs(coeff1234-
floatKCorrel(geneData,tfData,numDataCols,4,g+gpu,tf1,tf2,tf3,tf4,0))>=0.01)
{
#pragma omp critical
{
```

```
printf("MISMATCH from thread
%d!\n",thread_num);
printf("g=%d, tf1=%d,tf2=%d,tf3=%d,tf4=
%d\n",g+gpu,tf1,tf2,tf3,tf4);

printf("c1=%f\ta=
%f\n",coeff1,floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf1,0,0,0,0));
printf("c2=%f\ta=
%f\n",coeff2,floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf2,0,0,0,0));
printf("c3=%f\ta=
%f\n",coeff3,floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf3,0,0,0,0));
printf("c4=%f\ta=
%f\n",coeff4,floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,tf4,0,0,0,0));

printf("c12=%f\ta=
%f\n",coeff12,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf2,0,0,0));
printf("c13=%f\ta=
%f\n",coeff13,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf3,0,0,0));
printf("c14=%f\ta=
%f\n",coeff14,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf1,tf4,0,0,0));
printf("c23=%f\ta=
%f\n",coeff23,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf2,tf3,0,0,0));
printf("c24=%f\ta=
%f\n",coeff24,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf2,tf4,0,0,0));
printf("c34=%f\ta=
%f\n",coeff34,floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,tf3,tf4,0,0,0));

printf("c123=%f\t,a=
%f\n",coeff123,floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf2,tf3,0,0));
printf("c124=%f\t,a=
%f\n",coeff124,floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf2,tf4,0,0));
printf("c134=%f\t,a=
%f\n",coeff134,floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf1,tf3,tf4,0,0));
printf("c234=%f\t,a=
%f\n",coeff234,floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,tf2,tf3,tf4,0,0));
printf("c1234=%f\t,a=
%f\n",coeff1234,floatKCorrel(geneData,tfData,numDataCols,4,g+gpu,tf1,tf2,tf3,tf4,0));

tf1=1000;tf2=1000;tf3=1000;tf4=1000;
}
}

}//validation for correctness for dev/debug

//}//in this thread's realm of work
}//tf4
}//tf3
}//tf2
//x++;
} //tf1

}  //openMP

for(int talloc=0;talloc<4;talloc++)
        {
        if(gpuTF4m[talloc]!=NULL)
                {
                cudaFreeHost(gpuTF4m[talloc]);
                }
        else
                {
                printf("Couldn't free gputf4m[%d!]! was null!\n",talloc);
                }
        }



} //successful sync on stream
```

```
                                                  else
                                                            {
                                                            printf("Error in syncrhonizing with a stream for k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                            }
                                                  } //set device successful
                                        else
                                                  {
                                                  printf("ERROR in selecting a CUDA device for k=3 result retrieval with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                  }
                                        }  //for loop for getting results...


                                        //print status message
                                        for(int gpu=0;gpu<numGPUs;gpu++)
                                                  {
                                                  printf("Finished processing for gene g=%d/%d=%f...\n",g+gpu,numGenes,(((float)(g+gpu))/((float)(numGenes))));
                                                  }


                                        //if(g>=0){g=numGenes+1;}//  commented out so it runs to completion....


                                        }//for each gene (indexed by g)

                              }
                    else
                              {
                              printf("ERROR in memory allocation/initialzation! Not attempting any CUDA!\n");
                              }

                    //free the memories!
                    printf("Freeing memories...\n");
                    for(int gpu=0;gpu<numGPUs;gpu++)
                    //for(gpu=gpuStartOn;gpu<numGPUs; gpu=gpu+(stayOnGPU?1:0))
                              {

                              if(resultData022[gpu]!=NULL)
                                        cudaFreeHost(resultData022[gpu]);
                              if(resultData002[gpu]!=NULL)
                                        cudaFreeHost(resultData002[gpu]);
                              if(resultData222[gpu]!=NULL)
                                        cudaFreeHost(resultData222[gpu]);
                              if(resultData000[gpu]!=NULL)
                                        cudaFreeHost(resultData000[gpu]);
                              if(resultData_dev[gpu]!=NULL)
                                        cudaFree(resultData_dev[gpu]);
                              if(geneData_dev[gpu]!=NULL)
                                        cudaFree(geneData_dev[gpu]);
                              if(tfData_dev[gpu]!=NULL)
                                        cudaFree(tfData_dev[gpu]);
                              /*if(results000_dev[gpu]!=NULL)
                                        cudaFree(results000_dev[gpu]);
                              if(results002_dev[gpu]!=NULL)
                                        cudaFree(results002_dev[gpu]);
                              if(results022_dev[gpu]!=NULL)
                                        cudaFree(results022_dev[gpu]);
                              if(results222_dev[gpu]!=NULL)
                                        cudaFree(results222_dev[gpu]);
                              if(resultscomp_dev[gpu]!=NULL)
                                        cudaFree(resultscomp_dev[gpu]);
                              if(resultstf4_dev[gpu]!=NULL)
                                        cudaFree(resultstf4_dev[gpu]);*/
                              cudaStreamDestroy(cudaStreams[gpu]);
                              }
                    printf("Done freeing memories!  Returning!\n");

                    }////////////////////////////////////////////////K=4
```

```
void k3AnalysisMultiGPU(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int numGPUs)
          {

          /*
          alternate GPUs with each call to a kernel being a full k=3 analysis, but with
          a different gene.  use 3-dimensional blocks 10x10x10=1000 threads/block
          Use 3-D grids as well...that may speed things up too..
          */


          int gpu=0;
          float* resultData[2];
          float* resultData_dev[2];
          cudaStream_t cudaStreams[2];
          //long numCoeffsPerKernel=350*350*350;
          int  numCoeffsPerKernel=350*350*350;

          bool allocError=false;


          //compute TF indices
          int tf1;
          int tf2;
          int tf3;
          float* results;
          int gridx=0;int gridy=0;int gridz=0;
          int blockx=0;int blocky=0;int blockz=0;

          struct k3TFTriple tempTriple;

          //set up valid indices to evaluate returned results at
          vector<int> valid350By350By350Indices;
          vector<k3TFTriple> k3TFTuplesPreComputed;
          vector<int> indices12;
          vector<int> indices13;
          vector<int> indices23;
          vector<int> indices1;
          vector<int> indices2;
          vector<int> indices3;
          //printf("gx\tgy\tgz\tbx\tby\tbz\ttf1\ttf2\ttf3\tindex\t*\n");
          for(int x=0;x<350*350*350;x++)
                    {
                    //only increment block coordinates once we've moved
                    //to a new block
                    if((x%1000)==0 && x>0)
                              {
                              if(cycleInt(gridx,35))
                                        {
                                        if(cycleInt(gridy,35))
```

```
                                        {
                                        cycleInt(gridz,35);
                                        }
                              }
                    }
          //update block coordinates every time (except the first)
          if(x>0)
                    {
                    if(cycleInt(blockx,10))
                              {
                              if(cycleInt(blocky,10))
                                        {
                                        cycleInt(blockz,10);
                                        }
                              }
                    }
          tf1=gridx*10+blockx;
          tf2=gridy*10+blocky;
          tf3=gridz*10+blockz;
          //printf("gx\tgy\tgz\tbx\tgy\gz\ttf1\ttf2\ttf3\t*\tindex\n");
          //printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t",
          //                  gridx,gridy,gridz,blockx,blocky,blockz,tf1,tf2,tf3,x);
          if(tf1<tf2 && tf2<tf3)
                    {
                    //add to list of valid indices!
                    valid350By350By350Indices.push_back(x);
                    tempTriple.tf1=tf1;
                    tempTriple.tf2=tf2;
                    tempTriple.tf3=tf3;
                    k3TFTuplesPreComputed.push_back(tempTriple);
                    indices12.push_back(returnk2Index(tf1,tf2));
                    indices13.push_back(returnk2Index(tf1,tf3));
                    indices23.push_back(returnk2Index(tf2,tf3));
                    indices1.push_back(returnk2Index(tf1,tf1));
                    indices2.push_back(returnk2Index(tf2,tf2));
                    indices3.push_back(returnk2Index(tf3,tf3));
                    //printf("*");
                    }
          else
                    {
                    //printf("");
                    }


          //printf("\n");
          }

//printf("NUM valid indices : %ld\n",valid350By350By350Indices.size());

//set up triples that the GPU won't handle
vector<k3TFTriple> k3TFTuples;
for(tf1=0;tf1<numTFs-2;tf1++)
          {
          for(tf2=tf1+1;tf2<numTFs-2;tf2++)
                    {
                    for(tf3=tf2+1;tf3<numTFs;tf3++)
                              {
                              if(tf1<350 && tf2<350 && tf3<350)
                                        {
                                        //the GPU will handle this!
                                        }//GPU
                              else
                                        {
                                        //the CPU can handle this!
                                        tempTriple.tf1=tf1;
                                        tempTriple.tf2=tf2;
                                        tempTriple.tf3=tf3;
                                        k3TFTuples.push_back(tempTriple);
                                        }//CPU
                              }//for tf3
                    }//for tf2
          }//for tf1
//printf("Num computations for CPU : %ld \n",k3TFTuples.size());
```

```
for(gpu=0;gpu<numGPUs;gpu++)
          {
          geneData_dev[gpu]=NULL;
          tfData_dev[gpu]=NULL;

          resultData[gpu]=NULL;
          resultData_dev[gpu]=NULL;

          //set the proper device
          if(cudaSuccess==cudaSetDevice(gpu))
                    {
                    //STREAMS
                    //printf("gstream...\n");
                    if(cudaSuccess!=cudaStreamCreate(&(cudaStreams[gpu])))
                              {
                              printf("Error allocating a stream for gpu=%d!\n",gpu);
                              allocError=true;
                              }
                    else
                              {

                              }


                    //GENE DATA
                    //printf("cgd...\n");
                    if(cudaSuccess!=cudaMalloc((void**)(&(geneData_dev[gpu])),numGeneDataPoints*sizeof(float)))
                              {
                              printf("GPU %d had error in allocating device memory for gene data!\n",gpu);
                              geneData_dev[gpu]=NULL;
                              allocError=true;
                              }
                    else
                              {
                              //copy gene data to device
                              if(cudaSuccess!=cudaMemcpyAsync(geneData_dev[gpu],
                                                  geneData,numGeneDataPoints*sizeof(float),
                                                  cudaMemcpyHostToDevice,cudaStreams[gpu]))
                                        {
                                        printf("ERROR (g=%d) in copying gene data to device!\n",gpu);
                                        allocError=true;
                                        }
                              }

                    //TF DATA
                    //printf("tfd...\n");
                    if(cudaSuccess!=cudaMalloc((void**)(&(tfData_dev[gpu])),numTFDataPoints*sizeof(float)))
                              {
                              printf("ERROR (g=%d) allocating CUDA memory for TF data!\n",gpu);
                              tfData_dev[gpu]=NULL;
                              allocError=true;
                              }
                    else
                              {
                              //copy TF data to device
                              if(cudaSuccess!=cudaMemcpyAsync(tfData_dev[gpu],
                                                            tfData,numTFDataPoints*sizeof(float),
                                                            cudaMemcpyHostToDevice,
                                                            cudaStreams[gpu]))
                                        {
                                        printf("ERROR (g=%d) copying TF data to device memory!\n",gpu);
                                        allocError=true;
                                        }
                              }



                    //HOST data
```

```
                    //printf("hd...\n");
                    if(cudaSuccess!=cudaMallocHost((void**)&(resultData[gpu]),numCoeffsPerKernel*sizeof(float)))
                              {
                              printf("ERROR (g=%d) in allocating page-locked memory!!!\n",gpu);
                              resultData[gpu]=NULL;
                              allocError=true;
                              }
                    else
                              {
                              //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                              //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                              }

                    //GPU/device data
                    //printf("devd...\n");
                    if(cudaSuccess!=cudaMalloc((void**)(&(resultData_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                              {
                              allocError=true;
                              resultData_dev[gpu]=NULL;
                              printf("GPU %d had error in allocating device memory for results!\n",gpu);
                              }

                    }//set cuda device!
          else
                    {
                    allocError=true;
                    printf("Error in selecting CUDA device %d!\n",gpu);
                    }
          }//allocate memories, per host, per GPU (data, results, streams)




if(!allocError)
          {

          //10x10x10 blocks
          //35x35x35 grid!
          //Maximum sizes of each dimension of a block:    1024 x 1024 x 64
          //Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
          dim3 dimBlock(10,10,10);
          dim3 dimGrid(35,35,35);
          int geneIndex=0;
          for(int g=0;g<numGenes;g+=numGPUs)
                    {
                    printf("Doing a process loop g=%d,%d...\n",g,g+numGPUs-1);

                    for(int gpu=0;gpu<numGPUs;gpu++)
                              {
                              if(gpu<numGenes)
                                        {
                                        geneIndex=gpu+g;
                                        //printf("Now process gene=%d with gpu=%d!\n",gpu+g,gpu);
                                        if(cudaSetDevice(gpu)==cudaSuccess)
                                                  {
                                                  //call kernel (stream) for gene gpu+g (geneIndex)
                                                  gpu3kFunction<<<
                                                            dimGrid,dimBlock,0,cudaStreams[gpu]
                                                            >>>(
                                                            g+gpu,
                                                            geneData_dev[gpu],
                                                            tfData_dev[gpu],
                                                            numDataCols,
                                                            resultData_dev[gpu],
                                                            0,0,0
                                                            );
                                                  //copy back results (stream)
                                                  //printf("Going to copy back results...\n");
                                                  if(cudaSuccess!=cudaMemcpyAsync(
                                                            resultData[gpu],
                                                            resultData_dev[gpu],
                                                            numCoeffsPerKernel*sizeof(float),
```

```
                                        cudaMemcpyDeviceToHost,
                                        cudaStreams[gpu]))
                                        {
                                        printf("Error in copying back data from k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                        }

                                }
                        else
                                {
                                printf("ERROR in selecting a CUDA device for k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                }
                        }
                }

for(int gpu=0;gpu<numGPUs;gpu++)
                {
                //synchronize on stream
                //process results for gene g from gpu!
                if(cudaSetDevice(gpu)==cudaSuccess)
                        {
                        if(cudaStreamSynchronize(cudaStreams[gpu])==cudaSuccess)
                                {
                                results=resultData[gpu];

                                if(g+gpu<=1)
                                        {
                                        bool allGood=true;
                                        if(VALIDATE)
                                                {
                                                float* k3Grid=k3Analysis(geneData,geneNames,tfData,tfNames,
                                                        numGenes,numGeneDataPoints,numTFs,numTFDataPoints,numDataCols,
                                                        g+gpu,4);

                                                if(k3Grid!=NULL)
                                                        {
                                                        int maxve=350*350*350;
                                                        int k3m=0;int k3mm=0;
                                                        for(int ve=0;ve<maxve;ve++)
                                                                {
                                                                if(k3Grid[ve]!=(-18.0))
                                                                        {
                                                                        if(abs(k3Grid[ve]-results[ve])<=0.01)
                                                                                {
                                                                                k3m++;
                                                                                }
                                                                        else
                                                                                {
                                                                                k3mm++;
                                                                                }
                                                                        }
                                                                }
                                                        printf("Comparision results :\n");
                                                        printf("Match     :\t%d\n",k3m);
                                                        printf("MisMatch  :\t%d\n",k3mm);
                                                        free(k3Grid);
                                                        }
                                                }//VALIDATE


                                        #pragma omp parallel num_threads(4) default(shared)
                                                {

                                                //use valid indices to EVALUATE returned gpu data
                                                float gCoeff123,gCoeff12,gCoeff13,gCoeff23,gCoeff1,gCoeff2,gCoeff3;
                                                float coeff123,coeff12,coeff13,coeff23,coeff1,coeff2,coeff3;
                                                int num_threads=omp_get_num_threads();
                                                int thread_num=omp_get_thread_num();
                                                struct k3TFTriple coeffTuple;
                                                int tf1,tf2,tf3;
                                                float absImp;
                                                for(int i=0;i<valid350By350By350Indices.size();i++)
                                                        {
                                                        if(i%num_threads==thread_num)
```

```
{
//k=3
gCoeff123=results[valid350By350By350Indices[i]];

//k=2
gCoeff12=results[indices12[i]];
gCoeff13=results[indices13[i]];
gCoeff23=results[indices23[i]];

//k=1
gCoeff1=results[indices1[i]];
gCoeff2=results[indices2[i]];
gCoeff3=results[indices3[i]];

//look for co-factors using absolute improvement
absImp=calcK3AbsImp(
          gCoeff123,gCoeff12,gCoeff13,gCoeff23,
          gCoeff1,gCoeff2,gCoeff3);

coeffTuple=k3TFTuplesPreComputed[i];
tf1=coeffTuple.tf1;tf2=coeffTuple.tf2;tf3=coeffTuple.tf3;

if(returnk3Index(coeffTuple.tf1,coeffTuple.tf2,coeffTuple.tf3)!
=valid350By350By350Indices[i])
          {
          //BAD, they should equal
          //printf("index mismatch!\n");
          }
else
          {
          //good
          //if(i<=10){printf("good!\n");}
          }


//abs imp comp for GPU
if(absImp>=ABS_K3_IMP_THRESHOLD)
          {
          #pragma omp critical
                    {

          printf("GG\tGT1\tGT2\tGT3\tGC123\tGC12\tGC13\tGC23\tGC1\tGC2\tGC3\tGA\tGGN\tGT1N\tGT2N\tGT3N");
                    printf("%d\t%d\t%d\t%d\t",
                              g+gpu,tf1,tf2,tf3);//gene and TF ids
                    printf("%f\t",gCoeff123);//c123
                    printf("%f\t%f\t%f\t",gCoeff12,gCoeff13,gCoeff23);
                    printf("%f\t%f\t%f\t",gCoeff1,gCoeff2,gCoeff3);
                    printf("%f\t",absImp);
                    printf("%s\t%s\t%s\t
%s\n",geneNames[g+gpu],tfNames[tf1],tfNames[tf2],tfNames[tf3]);
                    }
          }



if(VALIDATE)
          {
          coeffTuple=k3TFTuplesPreComputed[i];

          //k=3

          coeff123=floatKCorrel(geneData,tfData,numDataCols,3,g+gpu,coeffTuple.tf1,coeffTuple.tf2,coeffTuple.tf3,0,0);

          //k=2

          coeff12=floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,coeffTuple.tf1,coeffTuple.tf2,0,0,0);

          coeff13=floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,coeffTuple.tf1,coeffTuple.tf3,0,0,0);

          coeff23=floatKCorrel(geneData,tfData,numDataCols,2,g+gpu,coeffTuple.tf2,coeffTuple.tf3,0,0,0);

          //k=1
          coeff1=floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,coeffTuple.tf1,0,0,0,0);
```

```
coeff2=floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,coeffTuple.tf2,0,0,0,0);
coeff3=floatKCorrel(geneData,tfData,numDataCols,1,g+gpu,coeffTuple.tf3,0,0,0,0);




if(
         abs(coeff123-gCoeff123)>0.01 ||
         abs(coeff12-gCoeff12)>0.01 ||
         abs(coeff13-gCoeff13)>0.01 ||
         abs(coeff23-gCoeff23)>0.01 ||
         abs(coeff1-gCoeff1)>0.01 ||
         abs(coeff2-gCoeff2)>0.01 ||
         abs(coeff3-gCoeff3)>0.01
         )
         {
         allGood=false;
         printf("Mismatch at i=%d\n",i);
         printf("tf1=%d\ttf2=%d\ttf3=

         printf("g123=%f\tg12=%f\tg13=%f\tg23=%f\tg1=%f\tg2=%f\tg3=

         printf("c123=%f\tc12=%f\tc13=%f\tc23=%f\tc1=%f\tc2=%f\tc3=

         printf("\n\n\n");
         }
else
         {
         /*printf("MATCH at i=%d\n",i);
         printf("tf1=%d\ttf2=%d\ttf3=

         printf("g123=%f\tg12=%f\tg13=%f\tg23=

         printf("c123=%f\tc12=%f\tc13=%f\tc23=

         printf("\n\n\n");*/
         }
}  //conditional validation of correctness



%d\n",coeffTuple.tf1,coeffTuple.tf2,coeffTuple.tf3);

%g\n",gCoeff123,gCoeff12,gCoeff13,gCoeff23,gCoeff1,gCoeff2,gCoeff3);

%g\n",coeff123,coeff12,coeff13,coeff23,coeff1,coeff2,coeff3);




%d\n",coeffTuple.tf1,coeffTuple.tf2,coeffTuple.tf3);

%f\n",gCoeff123,gCoeff12,gCoeff13,gCoeff23);

%f\n",coeff123,coeff12,coeff13,coeff23);


                                  //i=valid350By350By350Indices.size();
                                  if(!allGood)i=valid350By350By350Indices.size();
                                  }//each of the threads handles just its modn equivalent records

                         }//for valid indices
                }//main openMP block to evaluate GPU results returned


if(!allGood)
         {
         printf("ERROR! MISMATCH(ES)!\n");
         }
//else printf("ALL GOOD for gene=%d :)\n",g+gpu);

/*this openMP code performs analysis with TFs
350, 351, and 352.*/
#pragma omp parallel num_threads(4) default(shared)
         {
         struct k3TFTriple tempTriple;
         int size=k3TFTuples.size();
         int num_threads=omp_get_num_threads();
         int thread_num=omp_get_thread_num();
         float coeff123,coeff12,coeff13,coeff23,coeff1,coeff2,coeff3;
         float absImp;
         for(int c=0;c<size;c++)
                  {
                  if((c%num_threads)==thread_num)
                           {
                           //work on a job
```

```
                                                                                tempTriple=k3TFTuples[c];

                                                                                //k=3
                                                                                coeff123=floatKCorrel(geneData,tfData,numDataCols,3,
                                                                                          g+gpu,tempTriple.tf1,tempTriple.tf2,tempTriple.tf3,0,0);

                                                                                //k=2
                                                                                coeff12=floatKCorrel(geneData,tfData,numDataCols,2,
                                                                                          g+gpu,tempTriple.tf1,tempTriple.tf2,0,0,0);
                                                                                coeff13=floatKCorrel(geneData,tfData,numDataCols,2,
                                                                                          g+gpu,tempTriple.tf1,tempTriple.tf3,0,0,0);
                                                                                coeff23=floatKCorrel(geneData,tfData,numDataCols,2,
                                                                                          g+gpu,tempTriple.tf2,tempTriple.tf3,0,0,0);

                                                                                //k=1
                                                                                coeff1=floatKCorrel(geneData,tfData,numDataCols,1,
                                                                                          g+gpu,tempTriple.tf1,0,0,0,0);

                                                                                coeff2=floatKCorrel(geneData,tfData,numDataCols,1,
                                                                                          g+gpu,tempTriple.tf2,0,0,0,0);

                                                                                coeff3=floatKCorrel(geneData,tfData,numDataCols,1,
                                                                                          g+gpu,tempTriple.tf3,0,0,0,0);

                                                                                //compute the absolute improvement
                                                                                absImp=calcK3AbsImp(
                                                                                          coeff123,coeff12,coeff13,coeff23,
                                                                                          coeff1,coeff2,coeff3);

                                                                                //absImp comp for CPU
                                                                                if(absImp>=ABS_K3_IMP_THRESHOLD)
                                                                                          {
                                                                                          #pragma omp critical
                                                                                                    {

          printf("GG\tGT1\tGT2\tGT3\tGC123\tGC12\tGC13\tGC23\tGC1\tGC2\tGC3\tGA\tGGN\tGT1N\tGT2N\tGT3N");
                                                                                                                                                  printf("%d\t%d\t%d\t%d\t",
                                                                                                                                                            g+gpu,tempTriple.tf1,tempTriple.tf2,tempTriple.tf3);//gene
and TF ids
                                                                                                                                                  printf("%f\t",coeff123);//c123
                                                                                                                                                  printf("%f\t%f\t%f\t",coeff12,coeff13,coeff23);
                                                                                                                                                  printf("%f\t%f\t%f\t",coeff1,coeff2,coeff3);
                                                                                                                                                  printf("%f\t",absImp);
                                                                                                                                                  printf("%s\t%s\t%s\t
%s\n",geneNames[g+gpu],tfNames[tempTriple.tf1],tfNames[tempTriple.tf2],tfNames[tempTriple.tf3]);
                                                                                                                                                  }
                                                                                                    }
                                                                                }
                                                                                                                  }
                                                                                                        }//openMP


                                                                                          }//for g<=1, show results

                                                                                }//successful sync on stream
                                                                      else
                                                                                {
                                                                                printf("Error in syncrhonizing with a stream for k=3 analysis with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                                                }
                                                                      }//set device successful
                                                            else
                                                                      {
                                                                      printf("ERROR in selecting a CUDA device for k=3 result retrieval with gene=%d, gpu=%d!\n",geneIndex,gpu);
                                                                      }
                                                            }//for loop for getting results...


                                                  for(int gpu=0;gpu<numGPUs;gpu++)
                                                            {
                                                            printf("Finished processing for gene g=%d...\n",g+gpu);
                                                            }

                                                  //if(g>=0){g=numGenes;}
```

```
                        }//for each gene (indexed by g)

                }
        else
                {
                printf("ERROR in memory allocation/initialzation! Not attempting any CUDA!\n");
                }

        //free the memories!
        printf("Freeing memories...\n");
        for(int gpu=0;gpu<numGPUs;gpu++)
                {
                if(resultData[gpu]!=NULL)
                        cudaFree(resultData[gpu]);
                if(resultData_dev[gpu]!=NULL)
                        cudaFree(resultData_dev[gpu]);
                if(geneData_dev[gpu]!=NULL)
                        cudaFree(geneData_dev[gpu]);
                if(tfData_dev[gpu]!=NULL)
                        cudaFree(tfData_dev[gpu]);
                cudaStreamDestroy(cudaStreams[gpu]);
                }
        printf("Done freeing memories!  Returning!\n");

        }


inline float calcK3AbsImp(float c123,float c12,float c13,float c23,float c1,float c2,float c3)
        {

        float absImp=min(abs(c123-c12),min(
                        abs(c123-c13),min(
                        abs(c123-c23),min(
                        abs(c123-c1),min(
                        abs(c123-c2),
                        abs(c123-c3))))));

        return absImp;
        }


__global__ void gpu4kFunction(
        int geneIndex,
        float* geneData_dev,
        float* tfData_dev,
        int numPoints,
        float* res_dev,
        int tf1base,int tf2base,int tf3base,
        int tf1Hold
        ) {

        //much of the grid can remain uncomputed because of
        //commutativity of TF-co/tri-factors (min)
        if((blockIdx.y<blockIdx.x) || (blockIdx.z<blockIdx.y))
                {
                return;
                }

        //compute TF indices
        int tf1=(blockIdx.x*blockDim.x+threadIdx.x)+tf1base;
        int tf2=(blockIdx.y*blockDim.y+threadIdx.y)+tf2base;
        int tf3=(blockIdx.z*blockDim.z+threadIdx.z)+tf3base;

        //compute threadID in a block index (idx), blockId in a grid (bdx), and threadID overall (idx)
        int idx=threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.x*blockDim.y;  //threadID in a block (0-999)
        int bdx=blockIdx.x +blockIdx.y *gridDim.x +blockIdx.z*gridDim.x *gridDim.y;    //blockID in a grid;  (0-35^3)
        idx=(blockDim.x*blockDim.y*blockDim.z)*bdx+idx;

        //compute a coefficient!
        singleCUDAk4CorrelDEVICE(
```

```
                    //&(geneData_dev_s[0]),
                    &(geneData_dev[geneIndex*numPoints]),
                    &(tfData_dev[tf1*numPoints]),
                    &(tfData_dev[tf2*numPoints]),
                    &(tfData_dev[tf3*numPoints]),
                    &(tfData_dev[tf1Hold*numPoints]),
                    numPoints,
                    &(res_dev[idx]));

          }

__device__ void singleCUDAk4CorrelDEVICE(
          float* point1,
          float* point2,
          float* point3,
          float* point4,
          float* point5,
          int numPoints,
          float* res)
          {
          //point 1 reprsents base address of a single row of gene data
          //point 2 represents base address of a single row of TF data
          //point 3 represents base address of a single row of TF data
          //point 4 represents base address of a single row of TF data
          //point 5 represents base address of a single row of TF data
          //numPoints represent the number of data points in a particular row
          float xy=0;float x=0;float y=0;float xs=0;float ys=0;
          int numEffective=0;
          float coeff;
          float tempMin;
          bool isValidPoint=true;

          for(int i=0;i<numPoints;i++) {
                    //eliminate an if-statement with this 'isValidPoint' boolean variable!
                    isValidPoint=(point1[i]!=(-18) && point2[i]!=(-18) && point3[i]!=(-18) && point4[i]!=(-18) && point5[i]!=(-18));
                    tempMin=min(point2[i],min(point3[i],min(point4[i],point5[i])));
                    xy=xy+(isValidPoint)*(point1[i]*tempMin);
                    x=x+(isValidPoint)*point1[i];
                    y=y+(isValidPoint)*tempMin;
                    xs=xs+(isValidPoint)*(point1[i]*point1[i]);
                    ys=ys+(isValidPoint)*(tempMin*tempMin);
                    numEffective=numEffective+(isValidPoint*1);
                    }
          float num=(float)(numEffective)*xy-x*y;
          float denom=sqrtf(
                              ((float)(numEffective)*xs-x*x)
                              *
                              ((float)(numEffective)*ys-y*y)
                              );
          coeff=num/denom;
          *res=coeff;
          //*res=point1[0];//+(point2[0]/100.0f);
          }




__global__ void gpu3kFunction(
          int geneIndex,
          float* geneData_dev,
          float* tfData_dev,
          int numPoints,
          float* res_dev,
          int tf1base,int tf2base,int tf3base
          ) {

          //much of the grid can remain uncomputed because of
          //commutativity of TF-co/tri-factors (min)
          if((blockIdx.y<blockIdx.x) || (blockIdx.z<blockIdx.y))
                    {
```

```
                    return;
                    }

          //compute TF indices
          int tf1=(blockIdx.x*blockDim.x+threadIdx.x)+tf1base;
          int tf2=(blockIdx.y*blockDim.y+threadIdx.y)+tf2base;
          int tf3=(blockIdx.z*blockDim.z+threadIdx.z)+tf3base;

          //compute threadID in a block index (idx), blockId in a grid (bdx), and threadID overall (idx)
          int idx=threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.x*blockDim.y;  //threadID in a block (0-999)
          int bdx=blockIdx.x +blockIdx.y *gridDim.x +blockIdx.z*gridDim.x *gridDim.y;    //blockID in a grid;  (0-35^3)
          idx=(blockDim.x*blockDim.y*blockDim.z)*bdx+idx;

          //compute a coefficient!
          singleCUDAk3CorrelDEVICE(
                    //&(geneData_dev_s[0]),
                    &(geneData_dev[geneIndex*numPoints]),
                    &(tfData_dev[tf1*numPoints]),
                    &(tfData_dev[tf2*numPoints]),
                    &(tfData_dev[tf3*numPoints]),
                    numPoints,
                    &(res_dev[idx]));
          }


__device__ void singleCUDAk3CorrelDEVICE(
          float* point1,
          float* point2,
          float* point3,
          float* point4,
          int numPoints,
          float* res)
          {
          //point 1 reprsents base address of a single row of gene data
          //point 2 represents base address of a single row of TF data
          //point 3 represents base address of a single row of TF data
          //point 4 represents base address of a single row of TF data
          //numPoints represent the number of data points in a particular row
          float xy=0;float x=0;float y=0;float xs=0;float ys=0;
          int numEffective=0;
          float coeff;
          float tempMin;
          bool isValidPoint=true;


          for(int i=0;i<numPoints;i++) {
                    //eliminate an if-statement with this 'isValidPoint' boolean variable!
                    isValidPoint=(point1[i]!=(-18) && point2[i]!=(-18) && point3[i]!=(-18) && point4[i]!=(-18));
                    tempMin=min(point2[i],min(point3[i],point4[i]));
                    xy=xy+(isValidPoint)*(point1[i]*tempMin);
                    x=x+(isValidPoint)*point1[i];
                    y=y+(isValidPoint)*tempMin;
                    xs=xs+(isValidPoint)*(point1[i]*point1[i]);
                    ys=ys+(isValidPoint)*(tempMin*tempMin);
                    numEffective=numEffective+(isValidPoint*1);
                    }
          float num=(float)(numEffective)*xy-x*y;
          float denom=sqrtf(
                              ((float)(numEffective)*xs-x*x)
                              *
                              ((float)(numEffective)*ys-y*y)
                              );
          coeff=num/denom;
          *res=coeff;
          //*res=point1[0];//+(point2[0]/100.0f);
          }




float* k2AnalysisMultiGPU(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int numGPUs,
```

```
int numCoeffsPerKernel,dim3 dg,dim3 db)
          {

          /**
          /home/esalina/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/deviceQuery Starting...
           CUDA Device Query (Runtime API) version (CUDART static linking)
          Found 2 CUDA Capable device(s)
          Device 0: "GeForce GTX 590"
            CUDA Driver Version / Runtime Version          4.0 / 4.0
            CUDA Capability Major/Minor version number:    2.0
            Total amount of global memory:                 1536 MBytes (1610285056 bytes)
            (16) Multiprocessors x (32) CUDA Cores/MP:     512 CUDA Cores
            Total amount of shared memory per block:       49152 bytes
            Total number of registers available per block: 32768
            Warp size:                                     32
            Maximum number of threads per block:           1024
            Maximum sizes of each dimension of a block:    1024 x 1024 x 64
            Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
          Device 1: "GeForce GTX 590"
            CUDA Driver Version / Runtime Version          4.0 / 4.0
            CUDA Capability Major/Minor version number:    2.0
            Total amount of global memory:                 1535 MBytes (1609760768 bytes)
            (16) Multiprocessors x (32) CUDA Cores/MP:     512 CUDA Cores
            Total amount of shared memory per block:       49152 bytes
            Total number of registers available per block: 32768
            Warp size:                                     32
            Maximum number of threads per block:           1024
            Maximum sizes of each dimension of a block:    1024 x 1024 x 64
            Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535

          NVIDIA CUDA C Programming Guide (v 4.0)
          When using the runtime API (Section 3.2), the
          execution configuration is specified by inserting
          an expression of the form <<< Dg, Db, Ns, S >>>
           between the function name and the parenthesized
           argument list, where:

               Dg is of type dim3 (see Section B.3.2) and specifies
          the dimension and size of the grid, such that
          Dg.x * Dg.y * Dg.z equals the number of blocks
          being launched; Dg.z must be equal to 1 for
          devices of compute capability 1.x;

               Db is of type dim3 (see Section B.3.2) and
          specifies the dimension and size of each block,
           such that Db.x * Db.y * Db.z equals the
          number of threads per block;

                    Ns is of type size_t and specifies the number
          of bytes in shared memory that is dynamically allocated
          per block for this call in addition to the statically
          allocated memory; this dynamically allocated memory is
          used by any of the variables declared as an external
          array as mentioned in Section B.2.3; Ns is an optional
          argument which defaults to 0;

          S is of type cudaStream_t and specifies the associated stream;
          S is an optional argument which defaults to 0.

          **/

          int numCoeffs=0;
          float* results=NULL;
          int* geneJobs=NULL;
          int* tf1Jobs=NULL;
          int* tf2Jobs=NULL;
          int jobIndex=0;
          int numJobs=0;
          int tf1;
          int tf2;
          int g;
          int tempIndex=0;
          int gpu=0;
```

```
bool memAllocFail[2];
int* geneJobData_dev[2];
int* tf1JobData_dev[2];
int* tf2JobData_dev[2];
float* resultData[2];
float* resultData_dev[2];
cudaStream_t cudaStreams[2];
float gpuVal=0.0;
float cpuVal=0.0;
long numComps=0;

//init jobs
geneJobs=(int*)(malloc(K2_NUMCOEFFS*sizeof(int)));
tf1Jobs=(int*)(malloc(K2_NUMCOEFFS*sizeof(int)));
tf2Jobs=(int*)(malloc(K2_NUMCOEFFS*sizeof(int)));

//abort if memory allocation failure
if(geneJobs==NULL || tf1Jobs==NULL || tf2Jobs==NULL)
          {
          printf("Failure to allocate job data!\n");
          condFree(geneJobs);
          condFree(tf1Jobs);
          condFree(tf2Jobs);
          return NULL;
          }

//init jobs
for(g=0;g<numGenes;g++)
          {
          for(tf1=0;tf1<numTFs-1;tf1++)
                    {
                    for(tf2=tf1+1;tf2<numTFs;tf2++)
                              {
                              if(numCoeffs%numCoeffsPerKernel==0)
                                        {
                                        //printf("Found a starting job j=%d, g=%d, tf1=%d, tf2=%d !\n",numJobs,g,tf1,tf2);
                                        geneJobs[numJobs]=g;
                                        tf1Jobs[numJobs]=tf1;
                                        tf2Jobs[numJobs]=tf2;
                                        numJobs++;
                                        //numCoeffs=0;
                                        }
                              numCoeffs++;
                              }
                    }
          }

printf("Number coefficients per job %d.\n",numCoeffsPerKernel);
printf("Number jobs : %d\n",numJobs);
jobIndex=0;

for(gpu=0;gpu<numGPUs;gpu++)
          {

          geneJobData_dev[gpu]=NULL;
          tf1JobData_dev[gpu]=NULL;
          tf2JobData_dev[gpu]=NULL;
          memAllocFail[gpu]=NULL;
          resultData[gpu]=NULL;
          resultData_dev[gpu]=NULL;

          //set the proper device
          if(cudaSuccess==cudaSetDevice(gpu))
                    {
                    //STREAMS
                    if(cudaSuccess!=cudaStreamCreate(&(cudaStreams[gpu])))
                              {
                              printf("Error allocating a stream for gpu=%d!\n",gpu);
                              memAllocFail[gpu]=true;
                              }
                    else
                              {
```

```
          }

//JOB DATA
if(cudaSuccess!=cudaMalloc((void**)(&(geneJobData_dev[gpu])),numCoeffsPerKernel*sizeof(int)))
          {
          geneJobData_dev[gpu]=NULL;
          printf("Error in cudamalloc for gene index job copy over!\n");
          memAllocFail[gpu]=true;
          }
if(cudaSuccess!=cudaMalloc((void**)(&(tf1JobData_dev[gpu])),numCoeffsPerKernel*sizeof(int)))
          {
          tf1JobData_dev[gpu]=NULL;
          printf("Error in cudamalloc for tf1 index job copy over!\n");
          memAllocFail[gpu]=true;
          }
if(cudaSuccess!=cudaMalloc((void**)(&(tf2JobData_dev[gpu])),numCoeffsPerKernel*sizeof(int)))
          {
          tf2JobData_dev[gpu]=NULL;
          printf("Error in cudamalloc for tf2 index job copy over!\n");
          memAllocFail[gpu]=true;
          }

//GENE DATA
if(cudaSuccess!=cudaMalloc((void**)(&(geneData_dev[gpu])),numGeneDataPoints*sizeof(float)))
          {
          printf("GPU %d had error in allocating device memory for gene data!\n",gpu);
          geneData_dev[gpu]=NULL;
          memAllocFail[gpu]=true;
          }
else
          {
          //copy gene data to device
          if(cudaSuccess!=cudaMemcpyAsync(geneData_dev[gpu],
                                        geneData,numGeneDataPoints*sizeof(float),
                                        cudaMemcpyHostToDevice,cudaStreams[gpu]))
                    {
                    printf("ERROR (g=%d) in copying gene data to device!\n",gpu);
                    memAllocFail[gpu]=true;
                    }
          }

//TF DATA
if(cudaSuccess!=cudaMalloc((void**)(&(tfData_dev[gpu])),numTFDataPoints*sizeof(float)))
          {
          printf("ERROR (g=%d) allocating CUDA memory for TF data!\n",gpu);
          tfData_dev[gpu]=NULL;
          memAllocFail[gpu]=true;
          }
else
          {
          //copy TF data to device
          if(cudaSuccess!=cudaMemcpyAsync(tfData_dev[gpu],
                                              tfData,numTFDataPoints*sizeof(float),
                                              cudaMemcpyHostToDevice,
                                              cudaStreams[gpu]))
                    {
                    printf("ERROR (g=%d) copying TF data to device memory!\n",gpu);
                    memAllocFail[gpu]=true;
                    }
          }

//FIRST BATCH OF JOBS
/*printf("GPU = %d\n",gpu);
printf("gindexptr = %p\n",geneJobData_dev[gpu]);
printf("tf1indexptr = %p\n",tf1JobData_dev[gpu]);
printf("tf2indexptr = %p\n",tf2JobData_dev[gpu]);*/
if(geneJobData_dev[gpu]!=NULL && tf1JobData_dev[gpu]!=NULL && tf2JobData_dev[gpu]!=NULL)
          {
          if(copyK2JobsToDev(
                    geneJobs[0+gpu],
                    tf1Jobs[0+gpu],
                    tf2Jobs[0+gpu],
                    numGenes,
```

```
                                        numTFs,
                                        geneJobData_dev[gpu],
                                        tf1JobData_dev[gpu],
                                        tf2JobData_dev[gpu],
                                        numCoeffsPerKernel,
                                        cudaStreams[gpu]
                                        ))
                                        {
                                        printf("ERROR IN COPY OVER!!!!\n");
                                        }
                                else
                                        {
                                        //printf("Apparent success in copy!\n");
                                        }
                                }
                        else
                                {
                                printf("No copying of first batch of jobs, because a device ptr is NULL!\n");
                                }


                        //HOST data
                        if(cudaSuccess!=cudaMallocHost((void**)&(resultData[gpu]),numCoeffsPerKernel*sizeof(float)))
                                {
                                printf("ERROR (g=%d) in allocating page-locked memory!!!\n",gpu);
                                resultData[gpu]=NULL;
                                memAllocFail[gpu]=true;
                                }
                        else
                                {
                                //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                                //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                                }

                        //GPU/device data
                        if(cudaSuccess!=cudaMalloc((void**)(&(resultData_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                                {
                                memAllocFail[gpu]=true;
                                resultData_dev[gpu]=NULL;
                                printf("GPU %d had error in allocating device memory for results!\n",gpu);
                                }

                        }//set cuda device!
                else
                        {
                        printf("Error in selecting CUDA device %d!\n",gpu);
                        }
                }//init GPU-specific resources


if(allFalse(memAllocFail,numGPUs))
                {
                for(jobIndex=0;jobIndex<numJobs;jobIndex+=numGPUs)
                        {
                        for(gpu=0;gpu<numGPUs;gpu++)
                                {
                                if(gpu+jobIndex<numJobs)
                                        {
                                        if(cudaSetDevice(gpu)==cudaSuccess)
                                                {
                                                //copy over jobs
                                                if(copyK2JobsToDev(
                                                        geneJobs[jobIndex+gpu],
                                                        tf1Jobs[jobIndex+gpu],
                                                        tf2Jobs[jobIndex+gpu],
                                                        numGenes,
                                                        numTFs,
                                                        geneJobData_dev[gpu],
                                                        tf1JobData_dev[gpu],
                                                        tf2JobData_dev[gpu],
                                                        numCoeffsPerKernel,
                                                        cudaStreams[gpu]
                                                        ))
```

```
                                        {
                                        printf("ERROR IN COPY OVER!!!!\n");
                                        }
                                else
                                        {
                                        //printf("Apparent success in copy!\n");
                                        }

                                //call cuda  (async & stream)
                                GPUk2TupleFunction<<<dg,db,0,cudaStreams[gpu]>>>(
                                        geneJobs[jobIndex+gpu],
                                        tf1Jobs[jobIndex+gpu],
                                        tf2Jobs[jobIndex+gpu],
                                        numCoeffsPerKernel,
                                        geneData_dev[gpu],
                                        tfData_dev[gpu],
                                        resultData_dev[gpu],
                                        geneJobData_dev[gpu],
                                        tf1JobData_dev[gpu],
                                        tf2JobData_dev[gpu],
                                        numDataCols
                                        );

                                //get results (async & stream)
                                if(cudaSuccess!=cudaMemcpyAsync(
                                        resultData[gpu],resultData_dev[gpu],
                                        numCoeffsPerKernel*sizeof(float),
                                        cudaMemcpyDeviceToHost,cudaStreams[gpu]))
                                        {
                                        printf("ERROR in copying CUDA results back! g=%d, job=%d!\n",gpu,jobIndex+gpu);
                                        }


                                }
                        else
                                {
                                printf("Error selecting a CUDA device=%d for job=%d!\n",gpu,jobIndex);
                                }//failure to select device
                        }//within range of jobs
                }//for each of the GPUs, RUN CUDA!

        //analyze the results back from the GPUs!
        for(gpu=0;gpu<numGPUs;gpu++)
                {
                if(gpu+jobIndex!=(numJobs))
                        {
                        //wait for results (stream sync)
                        //read results (they've already been copied)
                        if(cudaSuccess!=cudaSetDevice(gpu))
                                {
                                printf("Error selecting a CUDA device for sync! g=%d\n",gpu);
                                }
                        else
                                {
                                //successfully selected a device!
                                if(cudaSuccess!=cudaStreamSynchronize(cudaStreams[gpu]))
                                        {
                                        printf("Error on synchronization! g=%d\n",gpu);
                                        }
                                else
                                        {
                                        //set working point to copied back data
                                        results=resultData[gpu];
                                        tempIndex=0;
                                        if(jobIndex%10000==0)printf("Getting a batch of results job=%d/%d, gpu=%d\n",jobIndex+gpu,numJobs,gpu);
                                        g=geneJobs[jobIndex+gpu];
                                        tf1=tf1Jobs[jobIndex+gpu];
                                        tf2=tf2Jobs[jobIndex+gpu];

                                        while(tempIndex<numCoeffsPerKernel && g<3166 && tf1<351 && tf2<352)
                                                {
                                                //cpuVal=floatKCorrel(geneData,tfData,numDataCols,2,g,tf1,tf2,0,0,0);
                                                gpuVal=results[tempIndex];
```

```
                                                                        cpuVal=gpuVal;
                                                                        numComps++;
                                                                        if(abs(gpuVal-cpuVal)>=0.001)
                                                                                  {
                                                                                  printf("FOUND A DIFF!\t");
                                                                                  printf("CIJ=%d, GPU=%d, job=%d/%d, Gene=%d, tf1=%d, tf2=%d, GPU=%f, CPU=%f\n",
                                                                                            tempIndex,gpu,jobIndex+gpu,numJobs,
                                                                                            g,tf1,tf2,gpuVal,cpuVal);
                                                                                  }
                                                                        else
                                                                                  {
                                                                                  /*printf("FOUND A SIMILARITY!\t");
                                                                                  printf("CIJ=%d, GPU=%d, job=%d/%d, Gene=%d, tf1=%d, tf2=%d, GPU=%f, CPU=%f\n",
                                                                                            tempIndex,gpu,jobIndex+gpu,numJobs,
                                                                                            g,tf1,tf2,gpuVal,cpuVal);*/
                                                                                  }

                                                                        tf2++;
                                                                        if(tf2==352)
                                                                                  {
                                                                                  tf1++;
                                                                                  tf2=tf1+1;
                                                                                  }
                                                                        if(tf1==351)
                                                                                  {
                                                                                  g++;
                                                                                  tf1=0;
                                                                                  tf2=1;
                                                                                  }
                                                                        if(g==3166)
                                                                                  {
                                                                                  tempIndex=numCoeffsPerKernel;
                                                                                  }

                                                                        tempIndex++;
                                                                        }
                                                              tempIndex=0;
                                                              }
                                                    }//successfully selected a device
                                          }//don't let gpu+jobIndex be too high
                                }//for each gpu
                      }//loop on jobs
          }
else
          {
          printf("UNABLE TO DO CUDA!\n");
          }


//release resources
for(gpu=0;gpu<numGPUs;gpu++)
          {
          if(geneJobData_dev[gpu]!=NULL)
                    cudaFree(geneJobData_dev[gpu]);
          if(tf1JobData_dev[gpu]!=NULL)
                    cudaFree(tf1JobData_dev[gpu]);
          if(tf2JobData_dev[gpu]!=NULL)
                    cudaFree(tf2JobData_dev[gpu]);
          if(resultData_dev[gpu]!=NULL)
                    cudaFree(resultData_dev[gpu]);
          if(resultData[gpu]!=NULL)
                    cudaFreeHost(resultData[gpu]);
          if(geneData_dev[gpu]!=NULL)
                    cudaFree(geneData_dev[gpu]);
          if(tfData_dev[gpu]!=NULL)
                    cudaFree(tfData_dev[gpu]);
          cudaStreamDestroy(cudaStreams[gpu]);
          }
condFree(geneJobs);
condFree(tf1Jobs);
condFree(tf2Jobs);

printf("NUMBER COMPARISONS : %ld\n",numComps);
```

```
          return NULL;
          }


float* k2AnalysisMultiGPUBLOCK(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int numGPUs)
          {
          //float* results=NULL;
          //int tf1;
          //int tf2;
          //int tempIndex=0;
          int gpu=0;
          float* resultData[2];
          float* resultData_dev[2];
          cudaStream_t cudaStreams[2];
          //float cpuVal=0.0;
          //float gpuVal=0.0;
          int numCoeffsPerKernel=1024;
          bool workDone[2];
          workDone[0]=false;workDone[1]=false;
          long numCorrectGenuineCoeffs=0;

          for(gpu=0;gpu<numGPUs;gpu++)
                    {
                    geneData_dev[gpu]=NULL;
                    tfData_dev[gpu]=NULL;

                    resultData[gpu]=NULL;
                    resultData_dev[gpu]=NULL;

                    //set the proper device
                    if(cudaSuccess==cudaSetDevice(gpu))
                              {
                              //STREAMS
                              //printf("gstream...\n");
                              if(cudaSuccess!=cudaStreamCreate(&(cudaStreams[gpu])))
                                        {
                                        printf("Error allocating a stream for gpu=%d!\n",gpu);
                                        }
                              else
                                        {

                                        }


                              //GENE DATA
                              //printf("cgd...\n");
                              if(cudaSuccess!=cudaMalloc((void**)(&(geneData_dev[gpu])),numGeneDataPoints*sizeof(float)))
                                        {
                                        printf("GPU %d had error in allocating device memory for gene data!\n",gpu);
                                        geneData_dev[gpu]=NULL;

                                        }
                              else
                                        {
                                        //copy gene data to device
                                        if(cudaSuccess!=cudaMemcpyAsync(geneData_dev[gpu],
                                                                  geneData,numGeneDataPoints*sizeof(float),
                                                                  cudaMemcpyHostToDevice,cudaStreams[gpu]))
                                                  {
                                                  printf("ERROR (g=%d) in copying gene data to device!\n",gpu);

                                                  }
                                        }

                              //TF DATA
                              //printf("tfd...\n");
                              if(cudaSuccess!=cudaMalloc((void**)(&(tfData_dev[gpu])),numTFDataPoints*sizeof(float)))
                                        {
                                        printf("ERROR (g=%d) allocating CUDA memory for TF data!\n",gpu);
                                        tfData_dev[gpu]=NULL;
```

```
                    }
                else
                    {
                    //copy TF data to device
                    if(cudaSuccess!=cudaMemcpyAsync(tfData_dev[gpu],
                                                    tfData,numTFDataPoints*sizeof(float),
                                                    cudaMemcpyHostToDevice,
                                                    cudaStreams[gpu]))
                        {
                        printf("ERROR (g=%d) copying TF data to device memory!\n",gpu);

                        }
                    }



                //HOST data
                //printf("hd...\n");
                if(cudaSuccess!=cudaMallocHost((void**)&(resultData[gpu]),numCoeffsPerKernel*sizeof(float)))
                    {
                    printf("ERROR (g=%d) in allocating page-locked memory!!!\n",gpu);
                    resultData[gpu]=NULL;

                    }
                else
                    {
                    //printf("SUCCESS (t=%d) in allocating page-locked memory!\n",tid);
                    //printf("Thread %d has address %p for host result data.\n",tid,resultData);
                    }

                //GPU/device data
                //printf("devd...\n");
                if(cudaSuccess!=cudaMalloc((void**)(&(resultData_dev[gpu])),numCoeffsPerKernel*sizeof(float)))
                    {

                    resultData_dev[gpu]=NULL;
                    printf("GPU %d had error in allocating device memory for results!\n",gpu);
                    }

                }//set cuda device!
            else
                {
                printf("Error in selecting CUDA device %d!\n",gpu);
                }
            }//allocate memories, per host, per GPU (data, results, streams)

dim3 dg(1,1,1);//dim grid
dim3 db(32,32,1);//dim block
struct k2BlockTuple* workTuples=allocBlockTuples(2);
inck2BlockTuple(&(workTuples[1]),numGenes,numTFs,32);
long loop=0;
while(!workDone[0] && !workDone[1])
            {

            for(gpu=0;gpu<numGPUs;gpu++)
                {
                if(cudaSetDevice(gpu)==cudaSuccess)
                                {
                    GPUk2BlockTupleFunction<<<dg,db,120*sizeof(float),cudaStreams[gpu]>>>(
                                workTuples[gpu].geneIndex,
                                workTuples[gpu].tf1Index,
                                workTuples[gpu].tf2Index,
                                geneData_dev[gpu],
                                tfData_dev[gpu],
                                resultData_dev[gpu],
                                numDataCols
                                );
                                if(cudaSuccess!=cudaMemcpyAsync(resultData[gpu],
                                                    resultData_dev[gpu]
                                                    ,1024*sizeof(float),
                                                    cudaMemcpyDeviceToHost,
                                                    cudaStreams[gpu]))
```

```
                                        {
                                        //error
                                        printf("error on copy back!!\n");
                                        }
                                }
          else
                    {
                    printf("Failure to set device for g=%d!, loop=%ld\n",gpu,loop);
                    }

          }

for(gpu=0;gpu<numGPUs;gpu++)
          {
          if(cudaSetDevice(gpu)==cudaSuccess)
                    {
                    if(cudaSuccess!=cudaStreamSynchronize(cudaStreams[gpu]))
                              {
                              printf("Error on synchronization! g=%d\n",gpu);
                              }
                    else
                              {
                              //tempIndex=0;
                              //results=resultData[gpu];
                              /*for(tf1=workTuples[gpu].tf1Index;tf1<workTuples[gpu].tf1Index+32;tf1++)
                                        {
                                        for(tf2=workTuples[gpu].tf2Index;tf2<workTuples[gpu].tf2Index+32;tf2++)
                                                  {
                                                  gpuVal=results[tempIndex];
                                                  cpuVal=floatKCorrel(geneData,tfData,numDataCols,2,workTuples[gpu].geneIndex,tf1,tf2,0,0,0);
                                                  printf("For gene=%d,tf1=%d,tf2=%d,gpu=%d, result=%f, cpu=%f\t",
                                                            workTuples[gpu].geneIndex,tf1,tf2,gpu,gpuVal,cpuVal);
                                                  if(abs(cpuVal-results[tempIndex])>=0.01)
                                                            {
                                                            printf("DIFF!");
                                                            }
                                                  else if(tf1>tf2)
                                                            {
                                                            numCorrectGenuineCoeffs++;
                                                            }
                                                  printf("\n");
                                                  tempIndex++;
                                                  }
                                        }*/
                              }
                    }
          else
                    {
                    printf("Failure to set device for g=%d!, loop=%ld\n",gpu,loop);
                    }
          }

//increment the tuples
for(gpu=0;gpu<numGPUs;gpu++)
          {
          //inck2BlockTuple(&(workTuples[gpu]),numGenes,numTFs,32);
          //inck2BlockTuple(&(workTuples[gpu]),numGenes,numTFs,32);
          inck2BlockTupleToNextUpperArraySpot(&(workTuples[gpu]),numGenes,numTFs,32);
          inck2BlockTupleToNextUpperArraySpot(&(workTuples[gpu]),numGenes,numTFs,32);
          }
loop++;
if(loop%10000==0)
          {
          printf("Loop is %ld...\n",loop);
          }
if(isk2BlockTupleMaxedOut(&(workTuples[0]),numGenes,numTFs,32))
          {
          workDone[0]=true;
          }
if(isk2BlockTupleMaxedOut(&(workTuples[1]),numGenes,numTFs,32))
          {
          workDone[1]=true;
          }
```

```
                //workDone[1]=true;workDone[0]=true;
                }


        //release resources
        for(gpu=0;gpu<numGPUs;gpu++)
                {
                //printf("g=%d, a\n",gpu);
                if(resultData_dev[gpu]!=NULL)
                        cudaFree(resultData_dev[gpu]);
                //printf("g=%d, d\n",gpu);
                if(resultData[gpu]!=NULL)
                        cudaFreeHost(resultData[gpu]);
                //printf("g=%d, c\n",gpu);
                if(geneData_dev[gpu]!=NULL)
                        cudaFree(geneData_dev[gpu]);
                //printf("g=%d, d\n",gpu);
                if(tfData_dev[gpu]!=NULL)
                        cudaFree(tfData_dev[gpu]);
                cudaStreamDestroy(cudaStreams[gpu]);
                }

        printf("Num correct, genuine coeffs (non-duplicated, correct values) : %ld.\n",numCorrectGenuineCoeffs);

        return NULL;
        }



__global__ void GPUk2BlockTupleFunction(
        int g,
        int tf1Base,
        int tf2Base,
        float* geneData_dev,
        float* tfData_dev,
        float* res_dev,
        int numPoints
        )
        {
        int tf1=tf1Base+threadIdx.y;//row
        int tf2=tf2Base+threadIdx.x;//column
        int idx=threadIdx.y*32+threadIdx.x;


        //shared memory for gene data
        __shared__ float geneData_dev_s[115];
        geneData_dev_s[min(idx,114)]=geneData_dev[g*numPoints+min(idx,114)];







        //res_dev[idx]=idx*2;
        singleCUDAk2CorrelDEVICE(
                //&(geneData_dev[g*numPoints]),
                &(geneData_dev_s[0]),
                &(tfData_dev[tf1*numPoints]),
                &(tfData_dev[tf2*numPoints]),
                numPoints,
                &(res_dev[idx]));

        }



__device__ void singleCUDAk2CorrelDEVICE(
        float* point1,
        float* point2,
        float* point3,
        int numPoints,
        float* res)
```

```
          {
          //point 1 reprsents base address of a single row of gene data
          //point 2 represents base address of a single row of TF data
          //point 3 represents base address of a single row of TF data
          //numPoints represent the number of data points in a particular row
          float xy=0;float x=0;float y=0;float xs=0;float ys=0;
          int numEffective=0;
          float coeff;
          float tempMin;
          bool isValidPoint=true;




          for(int i=0;i<115;i++) {
                    //eliminate an if-statement with this 'isValidPoint' boolean variable!
                    isValidPoint=(point1[i]!=(-18) && point2[i]!=(-18) && point3[i]!=(-18));
                    tempMin=min(point2[i],point3[i]);
                    xy=xy+(isValidPoint)*(point1[i]*tempMin);
                    x=x+(isValidPoint)*point1[i];
                    y=y+(isValidPoint)*tempMin;
                    xs=xs+(isValidPoint)*(point1[i]*point1[i]);
                    ys=ys+(isValidPoint)*(tempMin*tempMin);
                    numEffective=numEffective+(isValidPoint*1);
                    }
          float num=(float)(numEffective)*xy-x*y;
          float denom=sqrtf(
                              ((float)(numEffective)*xs-x*x)
                              *
                              ((float)(numEffective)*ys-y*y)
                              );
          coeff=num/denom;
          *res=coeff;
          }



__global__ void GPUk2TupleFunction(
          int g,
          int tf1,
          int tf2,
          int numJobsPerKernel,
          float* geneData_dev,
          float* tfData_dev,
          float* res_dev,
          int* geneJobData_dev,
          int* tf1JobData_dev,
          int* tf2JobData_dev,
          int numPoints
          )
          {

          int idx=threadIdx.x;//0...1023
          int geneIndex=geneJobData_dev[idx];
          int tf1Index=tf1JobData_dev[idx];
          int tf2Index=tf2JobData_dev[idx];
          /*int geneIndex=0;
          int tf1Index=0;
          int tf2Index=1;*/

          singleCUDAk2CorrelDEVICE(
                    &(geneData_dev[geneIndex*numPoints]),
                    &(tfData_dev[tf1Index*numPoints]),
                    &(tfData_dev[tf2Index*numPoints]),
                    numPoints,
                    &(res_dev[idx]));

          }

bool copyK2JobsToDev(
```

```
int startGene,
int startTF1,
int startTF2,
int numGenes,
int numTFs,
int* gene_jobs_dev,
int* tf1_jobs_dev,
int* tf2_jobs_dev,
int numCoeffsPerKernel,
cudaStream_t cudaStream
)
{
//error/return code
bool hadError=false;

//allocate memory whose data/contents will be copied over
int* geneJobData=NULL;
int* tf1JobData=NULL;
int* tf2JobData=NULL;

//allocate and zero
geneJobData=(int*)(calloc(numCoeffsPerKernel,sizeof(int)));
tf1JobData=(int*)(calloc(numCoeffsPerKernel,sizeof(int)));
tf2JobData=(int*)(calloc(numCoeffsPerKernel,sizeof(int)));
struct k2Tuple* myTuple=allocTuples(1);

//working variables
int index=0;

if(
          geneJobData!=NULL &&
          tf1JobData!=NULL &&
          tf2JobData!=NULL &&
          myTuple!=NULL
          )
          {
          geneJobData[index]=startGene;
          myTuple->tf1=startTF1;
          myTuple->tf2=startTF2;
          while(startGene<numGenes && index<numCoeffsPerKernel)
                    {


                    //set the job data
                    geneJobData[index]=startGene;
                    tf1JobData[index]=myTuple->tf1;
                    tf2JobData[index]=myTuple->tf2;

                    /*printf("Logging a job g=%d, tf1=%d, tf2=%d, index=%d\n",
                              geneJobData[index],tf1JobData[index],tf2JobData[index],index);*/



                    //increment the TFs
                    inck2Tuple(myTuple,numTFs);

                    //increment the gene conditionally
                    //and loop back the tuple conditionally
                    if(myTuple->tf1==numTFs-2)
                              {
                              //set the job data
                              index++;
                              geneJobData[index]=startGene;
                              tf1JobData[index]=myTuple->tf1;
                              tf2JobData[index]=myTuple->tf2;
                              /*printf("Logging a job g=%d, tf1=%d, tf2=%d, index=%d\n",
                                        geneJobData[index],tf1JobData[index],tf2JobData[index],index);*/
                              startGene++;
                              myTuple->tf1=0;
                              myTuple->tf2=1;
                              index++;
                              }
                    else
```

```
                                        {
                                        //increment index
                                        index++;
                                        }
                              }//while doing work

                    //now, copy over the data to the device!
                    /*printf("gjd is %p.\n",gene_jobs_dev);
                    printf("tf1p is %p.\n",tf1_jobs_dev);
                    printf("tf2p is %p.\n",tf2_jobs_dev);*/

                    if(cudaMemcpyAsync(gene_jobs_dev,geneJobData,numCoeffsPerKernel*sizeof(int),cudaMemcpyHostToDevice,cudaStream)!=cudaSuccess)
                              {
                              printf("Error in gene index data copy!\n");
                              hadError=true;
                              }
                    if(cudaMemcpyAsync(tf1_jobs_dev,tf1JobData,numCoeffsPerKernel*sizeof(int),cudaMemcpyHostToDevice,cudaStream)!=cudaSuccess)
                              {
                              printf("Error in tf1 index data copy!\n");
                              hadError=true;
                              }
                    if(cudaMemcpyAsync(tf2_jobs_dev,tf2JobData,numCoeffsPerKernel*sizeof(int),cudaMemcpyHostToDevice,cudaStream)!=cudaSuccess)
                              {
                              printf("Error in tf2 index data copy!\n");
                              hadError=true;
                              }

                    }//able to allocate memory
          else
                    {
                    printf("Error in CALLOC!\n");
                    hadError=true;
                    }//error in memory allocation

          //deallocate
          condFree(geneJobData);
          condFree(tf1JobData);
          condFree(tf2JobData);
          condFree(myTuple);



          //return any error
          return hadError;
          }




bool allFalse(bool* flags,int n)
          {
          for(int x=0;x<n;x++)
                    {
                    if(flags[x]==true)
                              return false;
                    }
          return true;
          }




__device__ bool twoTuplesEqualGPU(struct k2Tuple tup1,struct k2Tuple tup2)
          {
          if(
                    tup1.tf1==tup2.tf1 &&
                    tup1.tf2==tup2.tf2
                    )
                    {
                    return true;
```

```
                }
        else
                {
                return false;
                }
        }



__device__  void inck2TupleGPU(struct k2Tuple* myTuple,int numTFs)
        {
        if(
                myTuple->tf1==numTFs-2 &&
                myTuple->tf2==numTFs-1
                )
                {
                //can't inc!
                return;
                }

        myTuple->tf2=myTuple->tf2+1;
        if(myTuple->tf2==numTFs)
                {
                myTuple->tf1=myTuple->tf1+1;
                myTuple->tf2=myTuple->tf1+1;
                }
        }




inline void condCUDAFree(void* myCudaPtr_dev)
        {
        if(myCudaPtr_dev!=NULL)
                {
                cudaFree(myCudaPtr_dev);
                }
        }



/*
A PLAIN correlation coefficient function
(DOES check for (-18) 'invalid' data points!)
*/
inline float floatCorrel(float* point1,float* point2,int numPoints) {
        float xy=0;float x=0;float y=0;float xs=0;float ys=0;
        int effectiveNumPoints=0;
        int i=0;
        for(i=0;i<numPoints;i++) {
                if(point1[i]!=(-18.0) && point2[i]!=(-18.0))
                        {
                        //printf("Incorporating %f and %f into a coeff!\n",point1[i],point2[i]);
                        xy+=(point1[i]*point2[i]);
                        x+=point1[i];
                        y+=point2[i];
                        xs+=(point1[i]*point1[i]);
                        ys+=(point2[i]*point2[i]);
                        effectiveNumPoints++;
                        }
                }

        float num=(float)(effectiveNumPoints)*xy-x*y;
        float denom=sqrtf(
                        ((float)(effectiveNumPoints)*xs-x*x)
                        *
                        ((float)(effectiveNumPoints)*ys-y*y)
                        );
        float coeff=num/denom;


        return coeff;
        }
```

```
float* k3Analysis(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int geneIndex,int numOpenMPThreads)
          {

          //make sure n_threads>=1
          numOpenMPThreads=abs(numOpenMPThreads);
          numOpenMPThreads=max(1,numOpenMPThreads);

          float* k3Grid=NULL;
          k3Grid=(float*)malloc(350*350*350*sizeof(float));

          if(k3Grid!=NULL)
                    {
                    #pragma omp parallel num_threads(numOpenMPThreads) default(shared)
                              {
                              int tid=omp_get_thread_num();
                              int num_threads=omp_get_num_threads();
                              int x=0;int idx;



                              for(int gz=0;gz<35;gz++)
                                        {
                                        for(int gy=0;gy<35;gy++)
                                                  {
                                                  for(int gx=0;gx<35;gx++)
                                                            {
                                                            for(int bz=0;bz<10;bz++)
                                                                      {
                                                                      for(int by=0;by<10;by++)
                                                                                {
                                                                                for(int bx=0;bx<10;bx++)
                                                                                          {
                                                                                          if(x%num_threads==tid)
                                                                                                    {
                                                                                                    //use idx and the k3Grid pointer to store the resulting coeff!!!
                                                                                                    k3Function(geneIndex,geneData,tfData,numDataCols,
                                                                                                              gx,gy,gz,bx,by,bz,k3Grid,&idx);
                                                                                                    }
                                                                                          x++;
                                                                                          }
                                                                                }
                                                                      }
                                                            }
                                                  }
                                        }
                              }//openMP
                    }//k3Grid nonnull
          else
                    {
                    //k3grid null!!!
                    printf("k3grid null!!!\n");
                    }


          /*#pragma omp parallel num_threads(numOpenMPThreads) default(shared)
                    {
                    int tid=omp_get_thread_num();
                    int num_threads=omp_get_num_threads();
                    int x=0;
                    float coeff123;
                    float coeff12,coeff13,coeff23;
                    float coeff1,coeff2,coeff3;


                    for(int tf1=0;tf1<numTFs-2;tf1++)
                              {

                              for(int tf2=tf1+1;tf2<numTFs-1;tf2++)
                                        {

                                        for(int tf3=tf2+1;tf3<numTFs;tf3++)
```

```
                                        {
                                        if(tf1>=350 || tf2>=350 || tf3>=350)
                                                  {
                                                  if(x%num_threads==tid)
                                                            {
                                                            //k=3
                                                            coeff123=floatKCorrel(geneData,tfData,numDataCols,3,geneIndex,tf1,tf2,tf3,0,0);

                                                            //k=2
                                                            coeff12=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf2,0,0,0);
                                                            coeff13=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf1,tf3,0,0,0);
                                                            coeff23=floatKCorrel(geneData,tfData,numDataCols,2,geneIndex,tf2,tf3,0,0,0);

                                                            //k=3
                                                            coeff1=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf1,0,0,0,0);
                                                            coeff2=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf2,0,0,0,0);
                                                            coeff3=floatKCorrel(geneData,tfData,numDataCols,1,geneIndex,tf3,0,0,0,0);
                                                            }
                                                  x++;
                                                  }
                                        }
                                  }
                        }
                  }//openMP*/

         //printf("k3 analysis returning k3grid=%p...\n",k3Grid);
         return k3Grid;
         }




inline void k3Function(int g,float* geneData,float* tfData,int numDataCols,
          int blockx,int blocky,int blockz,int tx,int ty,int tz,float* results,int* ri)
          {



          //compute TF indices
          int tf1=blockx*10+tx;
          int tf2=blocky*10+ty;
          int tf3=blockz*10+tz;

          //compute threadID in a block index (idx), blockId in a grid (bdx), and threadID overall (idx)
          int idx=tx+ty*10+tz*10*10;                        //threadID in a block (0-999)
          int bdx=blockx +blocky*35 +blockz*35*35;          //blockID in a grid;  (0-35^3)
          idx=(10*10*10)*bdx+idx;

          //much of the grid can remain uncomputed because of
          //commutativity of TF-co/tri-factors (min)
          if((blocky<blockx) || (blockz<blocky))
                    {
                    results[idx]=(-18.0);
                    return;
                    }

          //compute the coefficient and store it in the proper place
          results[idx]=floatKCorrel
                              (geneData,tfData,numDataCols,3,g,tf1,tf2,tf3,0,0);
          *ri=idx;
          }




float* k2Analysis(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numDataCols,int numOpenMPThreads)
          {
```

```
	//long numCoeffs=0;
	float* k2Results=NULL;
	/*printf("Running a simple k=1 analysis...\n");
	printf("NumGenes = %d\n",numGenes);
	printf("NumGene data points = %d \n",numGeneDataPoints);
	printf("NumTFs = %d \n",numTFs);
	printf("Num TF data points = %d \n",numTFDataPoints);
	printf("Num cols = %d \n",numCols);*/

	k2Results=(float*)malloc(numGenes*numTFs*(numTFs-1)*sizeof(float));
	if(k2Results==NULL)
		{
		printf("FATAL ERROR : unable to allocate memory for k2 results!\n");
		}
	else
		{
		printf("Successfully allocated %ld bytes of memory for k2 results!\n",numGenes*numTFs*(numTFs-1)*sizeof(float));

		#pragma omp parallel num_threads(numOpenMPThreads) default(shared)
			{
			int tid=omp_get_thread_num();
			int num_threads=omp_get_num_threads();
			int index=tid;
			for(int g=0;g<numGenes;g++)
				{
				for(int tf1=0;tf1<numTFs-1;tf1++)
					{
					for(int tf2=tf1+1+tid;tf2<numTFs;tf2+=num_threads)
						{
						//(float* point1,float* point2,int numPoints,int k,int g,int t1,int t2,int t3,int t4,int t5)
						k2Results[index]=floatKCorrel(
							geneData,
							tfData,
							numDataCols,
							2,
							g,
							tf1,
							tf2,
							0,0,0
							);
						index+=num_threads;
						}
					}
				}
			}//omp parallel block
		}
	return k2Results;
	}



void cFoscJunValidationAnalysis(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numTFs,int numTFDataPoints,int numDataCols,bool hbo)
	{

	int fosIndices[4];
	int junIndices[4];

	fosIndices[0]=61;fosIndices[1]=62;fosIndices[2]=63;fosIndices[3]=64;
	junIndices[0]=106;junIndices[1]=107;junIndices[2]=108;junIndices[3]=109;

	/*
	62:FOS
	63:FOS
	64:FOSB
	65:FOSL1
	107:JUN
	108:JUN
	109:JUNB
	110:JUNB
	*/

	int fosIndex;
```

```
	int junIndex;
	float coeff1,coeff2,coeffc;
	float imp;

	printf("g,tf1,tf2,c1,c2,cc,i\n");
	for(int g=0;g<numGenes;g++)
		{
		for(int tf1=0;tf1<4;tf1++)
			{
			for(int tf2=0;tf2<4;tf2++)
				{
				fosIndex=fosIndices[tf1];
				junIndex=junIndices[tf2];
				//printf("Now considering tf1=%d/%s, tf2=%d/%s....",fosIndex,tfNames[fosIndex],junIndex,tfNames[junIndex]);
				//floatKCorrel(float* point1,float* point2,int numPoints,int k,int g,int t1,int t2,int t3,int t4,int t5)
				if(hbo)
					{
					coeff1=floatKCorrelHOB(geneData,tfData,numDataCols,1,g,fosIndex,0,0,0,0);
					coeff2=floatKCorrelHOB(geneData,tfData,numDataCols,1,g,junIndex,0,0,0,0);
					coeffc=floatKCorrelHOB(geneData,tfData,numDataCols,2,g,fosIndex,junIndex,0,0,0);
					}
				else
					{
					coeff1=floatKCorrel(geneData,tfData,numDataCols,1,g,fosIndex,0,0,0,0);
					coeff2=floatKCorrel(geneData,tfData,numDataCols,1,g,junIndex,0,0,0,0);
					coeffc=floatKCorrel(geneData,tfData,numDataCols,2,g,fosIndex,junIndex,0,0,0);
					}
				imp=min(abs(coeff1-coeffc),abs(coeff2-coeffc));
				//printf("\tc1=%f\tc2=%f\tcc=%f\ti=%f\n",coeff1,coeff2,coeffc,imp);

				/*if(coeffc>coeff1 && coeffc>coeff2)
					{
					imp=min(abs(coeff1-coeffc),abs(coeff2-coeffc));
					}
				else if(coeffc<coeff1 && coeffc<coeff2)
					{
					imp=min(abs(coeff1-coeffc),abs(coeff2-coeffc));
					}
				else
					{
					imp=0.0;
					}*/

				printf("%s,%s,%s,%f,%f,%f,%f\n",geneNames[g],tfNames[fosIndex],tfNames[junIndex],
					coeff1,coeff2,coeffc,imp);
				}
			}
		}



	}

float* k1Analysis(float* geneData,char** geneNames,float* tfData,char** tfNames,
int numGenes,int numGeneDataPoints,int numTFs,int numTFDataPoints,int numCols,int numOpenMPThreads)
	{


	long numCoeffs=0;
	float* k1Results=NULL;
	//float coeff;

	/*printf("Running a simple k=1 analysis...\n");
	printf("NumGenes = %d\n",numGenes);
	printf("NumGene data points = %d \n",numGeneDataPoints);
	printf("NumTFs = %d \n",numTFs);
	printf("Num TF data points = %d \n",numTFDataPoints);
	printf("Num cols = %d \n",numCols);*/

	k1Results=(float*)malloc(numGenes*numTFs*sizeof(float));
	if(k1Results==NULL)
		{
		printf("ERROR allocating memory for k1 results!\n");
```

```
                    return NULL;
                    }
          #pragma omp parallel num_threads(numOpenMPThreads) default(shared)
                    {
                    int resIndex;
                    int tid=omp_get_thread_num();
                    int num_threads=omp_get_num_threads();
                    for(int g=0;g<numGenes;g++)
                              {
                              //printf("Working with g=%d (%s)...\n",g,geneNames[g]);
                              resIndex=tid+g*numTFs;
                              for(int t=tid;t<numTFs;t+=num_threads)
                                        {
                                        //if(resIndex<=10)
                                        //printf("Now analysing for gene g=%d (%s), tf=%d (%s) ...\n",g,geneNames[g],t,tfNames[t]);
                                        //coeff=floatCorrel(float* point1,float* point2,int numPoints)
                                        k1Results[resIndex]=floatCorrel
                                                  (
                                                  &(geneData[numCols*g]),
                                                  &(tfData[numCols*t]),
                                                  numCols
                                                  );
                                        resIndex+=num_threads;
                                        //if(resIndex<=10)
                                        //printf("\tCoeff=%f\n",k1Results[resIndex-1]);
                                        //printf("**************************\n*****************************\n*************************\n");
                                        }
                              }
                    }


          numCoeffs=numGenes*numTFs;
          printf("Done running a simple k=1 analysis!\n");
          printf("Tot number of bytes needed for such coeffs : %ld \n",(long)(numCoeffs*(long)(sizeof(float))));
          return k1Results;
          }



void readRowsAndCols(int* nrgene,int* nrtf,int* ncols)
          {
          printf("Reading row/column data...\n");

          //printf("How many rows in gene file (NOT including header row ) :");
          scanf("%d",nrgene);
          //printf("How many rows in TF file (NOT including header row ) :");
          scanf("%d",nrtf);
          //printf("How many colums in data (including column name) :");
          scanf("%d",ncols);



          }



//number of rows
//number of DATA columns (includes row names!)
//pointer to a data array
//pointer to an array of char* (containing names)
int readDataFromSTDIN(int numRows,int numCols,float* data,char** names)
          {

          //init variables
          #define READ_STDIN_BUFFSIZE 100
          char buffer[READ_STDIN_BUFFSIZE];
          int numRowsToRead=0;
          int numColsToRead=0;
          float tempFloat=0.0;
          int row=0;
          int index=0;
          int col=0;
```

```
	int tmp;


	//initialize variables;
	for(tmp=0;tmp<READ_STDIN_BUFFSIZE;tmp++)
		{
		buffer[tmp]=NULL;
		}
	numRowsToRead=numRows;
	numColsToRead=numCols;




	//printf("Reading in row data from stdin....char* data data ....\n");
	//read in some rows of data
	for(row=0;row<numRowsToRead;row++) {
		//col goes to 116 to account for column with text
		for(col=0;col<numColsToRead;col++)
			{
			if(col==0)
				{
				//read strings (gene/TF names)
				scanf("%s",buffer);
				memcpy(names[row],buffer,strlen(buffer));
				//copy from temp buffer to name memory
				}
			else
				{
				//read expression data
				scanf("%f",&tempFloat);
				data[index]=tempFloat;
				//copy to real data
				index++;
				}
			//printf("Just read row=%d, col=%d\n",row,col);
			}
		}


	return 0;
	}



//expression=expression*exp(alpha*expression)
void mulAlpha(float alpha,float* expression,int num)
	{
	int index=0;

	//simple transform
	for(index=0;index<num;index++)
		{
		if(expression[index]!=(-18.0))
			{
			//if something isn't (-18) transform it
			expression[index]=expression[index]*exp(alpha*expression[index]);
			}
		else
			{
			//if it IS (-18), leave it at that so it won't be used in later computations
			}
		}
	}




__global__ void CUDAk11WayBatch(float* geneData_dev,float* tfData_dev,int numDataCols,int geneStart,float* resultData_dev,bool* recompFlags_dev)
	{
/*
```

```
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
*/
          //declare vars
          //Basically threadIdx.x and threadIdx.y are the numbers associated with each thread within a block.
          //The blockIdx.x and blockIdx.y refers to the label associated with a block in a grid. You are allowed up to a
          //2-dimensional grid (allowing for blockIdx.x and blockIdx.y). Basically, the blockIdx.x variable
          //is similar to the thread index except it refers to the number associated with the block.
          //gridDim.x=10

          int idx = blockIdx.x * blockDim.x + threadIdx.x;
          int t=idx%352;
          int g=(floor((float)(idx)/(float)(352)));
          //int g=geneStart;
          g+=geneStart;
          float resBack=(-99.0);

          //compute a coefficient
          //SingleCUDACorrelDEVICE(&(geneData_dev[g*numDataCols]),&(tfData_dev[t*numDataCols]),numDataCols,&(resultData_dev[idx]));
          SingleCUDACorrelDEVICE(&(geneData_dev[g*numDataCols]),&(tfData_dev[t*numDataCols]),numDataCols,&resBack);
          //resultData_dev[idx]=idx;
          resultData_dev[idx]=resBack;

          if(recompFlags_dev!=NULL)
                    {
                    recompFlags_dev[idx]=(1==(idx%2));
                    }

          }







/*
pointer to one row of values
pointer to another
integer, representing the number of values in the row
return the correlation coefficient
*/
__device__ void SingleCUDACorrelDEVICE(float* point1,float* point2,int numPoints,float* res)
          {
          float xy=0;float x=0;float y=0;float xs=0;float ys=0;
          int i=0;
          int numEffective=0;
          float coeff;
          for(i=0;i<numPoints;i++) {
                    if(point1[i]!=(-18) && point2[i]!=(-18))
                              {
                              xy+=(point1[i]*point2[i]);
                              x+=point1[i];
                              y+=point2[i];
                              xs+=(point1[i]*point1[i]);
                              ys+=(point2[i]*point2[i]);
                              numEffective++;
                              }
                    }

          float num=(float)(numEffective)*xy-x*y;
          float denom=sqrtf(
                              ((float)(numEffective)*xs-x*x)
                              *
                              ((float)(numEffective)*ys-y*y)
                              );
          if(numEffective>0)
                    {
                    coeff=num/denom;
                    //coeff=point2[0];
                    }
```

```
	else
		{
		coeff=(-18.0);
		}
	*res=coeff;

	}







inline float floatKCorrelHOB(float* point1,float* point2,int numPoints,int k,int g,int t1,int t2,int t3,int t4,int t5)
	{
	//point 1 reprsents base address of ALL gene data (not a particular gene)
	//point 2 represents base address of ALL TF data (no particular TF(s))
	//numPoints represent the number of data points in a particular row

	float cofactor[numPoints];
	float coeff;

	k=min(k,5);
	k=max(1,k);
	int t1index,t2index,t3index,t4index,t5index;
	t1index=t1*numPoints;

	if(k>1)
		{
		t2index=t2*numPoints;
		t3index=t3*numPoints;
		t4index=t4*numPoints;
		t5index=t5*numPoints;

		for(int i=0;i<numPoints;i++)
			{
			if(k==2)
				cofactor[i]=min(point2[t1index+i],point2[t2index+i]);
			if(k==3)
				cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],point2[t3index+i]));
			if(k==4)
				cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],min(point2[t3index+i],point2[t4index+i])));
			if(k==5)
				cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],min(point2[t3index+i],min(point2[t4index+i],point2[t5index+i]))));
			if(!isHOBTissueIndex(i))
				cofactor[i]=(-18);
			}
		}//for k>1
	else
		{
		//k=1 just copy to cofactor
		memcpy(cofactor,&(point2[t1index]),numPoints*sizeof(float));
		}

	coeff=floatCorrel(&(point1[g*numPoints]),cofactor,numPoints);

	return coeff;
	}






inline float floatKCorrel(float* point1,float* point2,int numPoints,int k,int g,int t1,int t2,int t3,int t4,int t5)
	{
	//point 1 reprsents base address of ALL gene data (not a particular gene)
	//point 2 represents base address of ALL TF data (no particular TF(s))
	//numPoints represent the number of data points in a particular row

	float cofactor[numPoints];
```

```
float coeff;

k=min(k,5);
k=max(1,k);
int t1index,t2index,t3index,t4index,t5index;
t1index=t1*numPoints;

if(k>1)
          {
          t2index=t2*numPoints;
          t3index=t3*numPoints;
          t4index=t4*numPoints;
          t5index=t5*numPoints;

          for(int i=0;i<numPoints;i++)
                    {
                    if(k==2)
                              cofactor[i]=min(point2[t1index+i],point2[t2index+i]);
                    if(k==3)
                              cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],point2[t3index+i]));
                    if(k==4)
                              cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],min(point2[t3index+i],point2[t4index+i])));
                    if(k==5)
                              cofactor[i]=min(point2[t1index+i],min(point2[t2index+i],min(point2[t3index+i],min(point2[t4index+i],point2[t5index+i]))));
                    }
          }//for k>1
else
          {
          //k=1 just copy to cofactor
          memcpy(cofactor,&(point2[t1index]),numPoints*sizeof(float));
          }

coeff=floatCorrel(&(point1[g*numPoints]),cofactor,numPoints);

return coeff;
}
```