```
/* This code was writen by Christopher Baker and Timothee Bouhour during fall semester of 2012 at the University

* of Illinois at Urbana-Champaign. This is a parallel implementation of the hyperspectral facial recognition

* algorithm that we have developed for ECE445 along with Akshay Malik.

*

* This code is written with the Kepler architecture in mind.

*/


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <float.h>


#include "entry.h"

#include "matrix.h"


#define SUBM_DIM 3 // x and y dimensions of a submatrix

#define VEC_SIZE 30 // number of bands for each pixel, z dimension of a feature's matrix

#define DIM_FEAT 9 // x and y dimensions of a feature's matrix

#define FEAT_COUNT 5 // number of features being compared

#define NUM_KERNELS 5 //number of kernels executing at once


#define TGT_THDS_PER_BLK (256*2)

#define THD_PER_FEAT_COMP (SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM)

#define SHAREDMEM_BLK 49152 //memory available per block
```

```
#define FEAT_BLK_THDS (TGT_THDS_PER_BLK/(THD_PER_FEAT_COMP))

#define FEAT_BLK_MEM
(SHAREDMEM_BLK/((sizeof(float)*VEC_SIZE*SUBM_DIM*SUBM_DIM) +
SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM*sizeof(float)))

#define FEAT_BLK ((FEAT_BLK_THDS < FEAT_BLK_MEM) ? FEAT_BLK_THDS :
FEAT_BLK_MEM)


#define BLOCK_SIZE 256


//define MACROS

#define POS(col,row,dim,size) ((col + row*dim)*size)


//define the weights

#define W_HAIR 0

#define W_RCHEEK 3

#define W_LCHEEK 3

#define W_FHEAD 5

#define W_LIPS 1

#define W_SPATIAL 0



//allocate constant memory to work with for target feature data

__constant__ float targetFeature[VEC_SIZE*SUBM_DIM*SUBM_DIM];

__constant__ float prec_rc_c[VEC_SIZE*VEC_SIZE];

__constant__ float prec_lc_c[VEC_SIZE*VEC_SIZE];

__constant__ float prec_lips_c[VEC_SIZE*VEC_SIZE];

__constant__ float prec_fh_c[VEC_SIZE*VEC_SIZE];

__constant__ float prec_hair_c[VEC_SIZE*VEC_SIZE];
```

```
//global variables for main unction
int feature_offset[5];


/*************************************************************************************
************************************/
//REDUCTION KERNEL
__global__ void reduction(float * out, int * out_idx, float * in, int * in_idx,  unsigned int
size )
{
        __shared__ float psum[10*BLOCK_SIZE];
        __shared__ int index[10*BLOCK_SIZE];


          int tx = threadIdx.x;
          int block_start = 10*blockIdx.x*blockDim.x;


          if(block_start + tx < size)
          {
                psum[tx] = in[block_start + tx];
                index[tx] = in_idx[block_start + tx];
          }
          else
          {
                psum[tx] = -1;
                index[tx] = -1;
          }


          if(block_start + blockDim.x + tx < size)
```

```
{
    psum[blockDim.x + tx] = in[block_start + blockDim.x + tx];
    index[blockDim.x + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
    psum[blockDim.x + tx] = -1;
    index[blockDim.x + tx] = -1;
}

if(block_start + blockDim.x*2 + tx < size)
{
    psum[blockDim.x*2 + tx] = in[block_start + blockDim.x + tx];
    index[blockDim.x*2 + tx] = in_idx[block_start + blockDim.x*2 + tx];
}
else
{
    psum[blockDim.x*2 + tx] = -1;
    index[blockDim.x*2 + tx] = -1;
}

if(block_start + blockDim.x*3 + tx < size)
{
    psum[blockDim.x*3 + tx] = in[block_start + blockDim.x + tx];
    index[blockDim.x*3 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
```

```
{
        psum[blockDim.x*3 + tx] = -1;
        index[blockDim.x*3 + tx] = -1;
}

if(block_start + blockDim.x*4 + tx < size)
{
        psum[blockDim.x*4 + tx] = in[block_start + blockDim.x + tx];
        index[blockDim.x*4 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
        psum[blockDim.x*4 + tx] = -1;
        index[blockDim.x*4 + tx] = -1;
}

if(block_start + blockDim.x*5 + tx < size)
{
        psum[blockDim.x*5 + tx] = in[block_start + blockDim.x + tx];
        index[blockDim.x*5 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
        psum[blockDim.x*5 + tx] = -1;
        index[blockDim.x*5 + tx] = -1;
}
```

```
if(block_start + blockDim.x*6 + tx < size)
{
	psum[blockDim.x*6 + tx] = in[block_start + blockDim.x + tx];
	index[blockDim.x*6 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
	psum[blockDim.x*6 + tx] = -1;
	index[blockDim.x*6 + tx] = -1;
}

if(block_start + blockDim.x*7 + tx < size)
{
	psum[blockDim.x*7 + tx] = in[block_start + blockDim.x + tx];
	index[blockDim.x*7 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
	psum[blockDim.x*7 + tx] = -1;
	index[blockDim.x*7 + tx] = -1;
}

if(block_start + blockDim.x*8 + tx < size)
{
	psum[blockDim.x*8 + tx] = in[block_start + blockDim.x + tx];
	index[blockDim.x*8 + tx] = in_idx[block_start + blockDim.x + tx];
}
```

```
else
{
        psum[blockDim.x*8 + tx] = -1;
        index[blockDim.x*8 + tx] = -1;
}

if(block_start + blockDim.x*9 + tx < size)
{
        psum[blockDim.x*9 + tx] = in[block_start + blockDim.x + tx];
        index[blockDim.x*9 + tx] = in_idx[block_start + blockDim.x + tx];
}
else
{
        psum[blockDim.x*9 + tx] = -1;
        index[blockDim.x*9 + tx] = -1;
}

for(int step = BLOCK_SIZE; step >=1; step >>= 1)
{
   __syncthreads();
   int i, j;
   float temp;
   int idx_temp;
   if(tx < step)
   {
      for(i = 1; i< 10; i++)
          for(j = 0; j<10; j++)
```

```
                {
                    if(psum[tx+step*i] < psum[tx + step*j] && psum[tx+step*i] >=
0)
                    {
                       temp = psum[tx+step*j];
                            psum[tx+step*j] = psum[tx+ step*i];
                            psum[tx+step*i] = temp;

                            idx_temp = index[tx+step*j];
                            index[tx+step*j] = index[tx+ step*i];
                            index[tx+step*i] = idx_temp;
                            break;
                    }
                }
            }
        }

        if(tx < 5)
        {
            out[blockIdx.x*5 + tx] = psum[tx];
            out_idx[blockIdx.x*5 + tx] = index[tx];
        }

}
```

```
/**************************************************************************************
************************************/



//multiplies two matricies, returns a pointer to the result or null on failure

__device__ void matrix_mul(float * a, float * b, int wa, int ha, int wb, int hb, float*
result)

{

  int i, j, k;

  //for( i = 0; i< wb*hb; i++)

          //   {

               //*(diff + i) = *(a + i) - *(b + i);

            //     printf("precmatix of %d is: %f \n", i, *(b + i));

             //}

  for(i = 0; i < ha; i++)

     for(j = 0; j < wb; j++)

        *(result + POS(j,i,wa,1)) = 0;

                        //printf("result_1 of %d is: %f \n", i, *(result_1 + i));

  //int i;

  //printf("wa is %d, ha is %d, wb is %d, hb is %d. \n ", wa, ha, wb, hb);

  for(i = 0; i < ha; i++)

     for(j = 0; j < wa; j++)

        for(k = 0; k < hb; k++)

        {
```

```
            *(result + POS(j,i,wa, 1)) += ((*(a + POS(k,i,wa,1))) * (*(b + POS(j,k,wb,1))));
            //printf("Hello world! \n");
            //printf("result at i: %d, j: %d, k: %d is: %f \n", i, j, k, *(a) );
        }

  //for(i = 0; i<VEC_SIZE; i++)
   //              printf("result of %d is: %f \n", i, *(result + i));
}

//helper that preforms the matrix math of the mahalanobis distance
__device__ void handle_matricies(float * a, float * b, float * prec, int dim, float * temp)
{
        float result_1[VEC_SIZE];
        float result_2[1];
   //float * result_1 = matrix_mul(a, prec, dim, 1, dim, dim);
   //float * result_2 = matrix_mul(result_1, b, dim, 1, 1, dim);
        matrix_mul(a, prec, dim, 1, dim, dim, result_1);
        matrix_mul(result_1, b, dim, 1, 1, dim, result_2);
   float square = *result_2;
   //printf("Target matrix \n");
   //printmatrix(result_2, 1);
   //free(result_1);
   //free(result_2);
   //printf("square is: %f \n", square);

   *temp = sqrt(square);
```

```
}

// MAHALNOBIS DISTANCE
__device__ void mahadist(float * a, float * b, float * prec, int dim, float * temp)
{

   int i,l,m;

   float diff[VEC_SIZE]; //= (float*)malloc(dim * sizeof(float));

   //for( i = 0; i< dim*dim; i++)
          //{
              //*(diff + i) = *(a + i) - *(b + i);
           //   printf("precmatrix of %d is: %f \n", i, *(prec_rc_c + i));
          //}

   for( i = 0; i< dim; i++)
   {
      *(diff + i) = *(a + i) - *(b + i);
      //printf("diff of %d is: %f \n", i, *(diff + i));
   }
  // for(l = 0; l < VEC_SIZE; l++)
    //   for(m = 0; m < VEC_SIZE; m++)
          //printf("prec 1st value %f\n", (float)*(prec));


   handle_matricies(diff, diff, prec, dim, temp);
```

```
}


// PRINT MATRICES
/*__device__ void gpuprintmatrix(float * matrix, int dim) {
   int i, j;
   for(i=0;i<dim;i++)
   {
      for(j = 0; j<dim; j++)
      {
         printf("%.05f ", *(matrix + i*dim + j));
      }
      printf("\n");
   }
}*/


/*************************************************************************************
************************************/


__global__ void parallelawesomecakes(float * in_data, float * out_data, float * targetF,
int jay, int kay, int features_blk, int matrixdim, int vec_size, float * precision_matrix,
int kernelnum, int * jj, int * kk)
{
   int f,l,m;
   f = threadIdx.x;


   //printf("kernel number is %d, jvar is %d, kvar is %d, thread is %d \n",
kernelnum, jay, kay, threadIdx.x);
```

```
   //printf("kernel number %d, out_data address is: %p \n", kernelnum, out_data);
/*
   if(threadIdx.x == 0 && jay == 0 && kay == 0)
   {
         for(l = 0; l < VEC_SIZE; l++)
               for(m = 0; m < VEC_SIZE; m++)
                     if(!((float)*(precision_matrix + l + m*VEC_SIZE) > -
999999999999))
                           printf("prec x: %d y: %d value %f\n", l,m,
*(precision_matrix + l + m*VEC_SIZE));
   }
*/

   //allocate shared memory for submatricies of target and entries
   __shared__ float subm[((FEAT_BLK * VEC_SIZE * DIM_FEAT *
DIM_FEAT)/(SUBM_DIM*SUBM_DIM))]; //need to include in memory block
allocation calculations in main
   __shared__ float result[SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM *
FEAT_BLK];
   //__shared__ float tsubm[SUBM_DIM*SUBM_DIM*vec_size];


   // iterate over the database entries, create submatricies for each, and run
calculations with these submatricies
   //int f = threadIdx.x;
   int i, j, k;
   float temp_sum;

   if(f < features_blk*SUBM_DIM*SUBM_DIM)
```

```
{
        //printf("got inside first loop \n");
    //for each entry,
    //form the submatricies
    //form division index for creating sub-matrix vector averages
    int div_idx = matrixdim/SUBM_DIM;
    //printf("div_idx is: %d \n", div_idx);



    //printf("entered f loop \n");
    // average the sub-matricies
    int ths = f%(SUBM_DIM*SUBM_DIM);
    i = ths%SUBM_DIM;
    j = (ths/SUBM_DIM)%SUBM_DIM;
    //i = ((f%features_blk)/SUBM_DIM) % SUBM_DIM;
    //j = (f%features_blk)%SUBM_DIM;
    //printf("first loop checkpoint \n");
    for( k = 0; k< VEC_SIZE; k++)
    {
        temp_sum = 0;
        for(l = 0; l < div_idx; l++)
                for(m = 0; m < div_idx; m++)
                {
                        temp_sum += *(in_data +
(f/(SUBM_DIM*SUBM_DIM))*VEC_SIZE*matrixdim*matrixdim +
i*VEC_SIZE*matrixdim*SUBM_DIM + j*VEC_SIZE*SUBM_DIM +
l*VEC_SIZE*matrixdim + m*VEC_SIZE + k);
                        //if(!(temp_sum > -9999999999999999))
```

```
                                        //printf("Houston we have a problem... \n");

                                    //if((POS(j,i,SUBM_DIM, 30) + k) ==89)

                                    //if(jay == 0 && kay == 0 && l==0 && m == 0 && k == 0)

                                        //printf("in_data[%d] = %f \n",
((f/(SUBM_DIM*SUBM_DIM))*VEC_SIZE*matrixdim*matrixdim + POS(l,m,
matrixdim, VEC_SIZE) + POS(i,j, SUBM_DIM, VEC_SIZE * div_idx) + k), *(in_data +
(f/(SUBM_DIM*SUBM_DIM))*VEC_SIZE*matrixdim*matrixdim + POS(l,m, matrixdim,
VEC_SIZE) + POS(i,j, SUBM_DIM, VEC_SIZE * div_idx) + k));

                                    //printf("sum of %d for thread %d \n" ,
j*VEC_SIZE*matrixdim*SUBM_DIM + i*VEC_SIZE*SUBM_DIM +
l*VEC_SIZE*matrixdim + m*VEC_SIZE + k, threadIdx.x);

                        }

          //printf("about to print to the subm array \n");

         subm[(f/(SUBM_DIM*SUBM_DIM))*VEC_SIZE*SUBM_DIM*SUBM_DIM +
POS(i,j, SUBM_DIM, VEC_SIZE) + k] = temp_sum/(div_idx*div_idx);

         //subm[f*SUBM_DIM*SUBM_DIM*vec_size + POS(i,j, SUBM_DIM, vec_size) + k]
= temp_sum/(div_idx*div_idx);

         //printf("loaded the number %f into subm at position %d \n",
temp_sum/(div_idx*div_idx),
(f/(SUBM_DIM*SUBM_DIM))*VEC_SIZE*SUBM_DIM*SUBM_DIM + POS(i,j,
SUBM_DIM, VEC_SIZE) + k);

      }

   }

   __syncthreads();

   /*if(threadIdx.x == 0)

   {

   for(i =0; i < SUBM_DIM; i++)

      for(j =0; j < SUBM_DIM; j++)

          for( k = 0; k< 30; k++)

             printf("Submatrix[%d] = %f \n", POS(i,j,SUBM_DIM, 30) + k,
subm[POS(i,j,SUBM_DIM, 30) + k]);

   }*/
```

```
  // call the mahalanobis distance on each
  if(f < features_blk*SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM)
  {
        //printf("got into second loop \n");
        int feature_num = f/(SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM);
        int threads = f%(SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM);
     i = threads%SUBM_DIM;
     j = (threads/SUBM_DIM)%SUBM_DIM;
     k = (threads/(SUBM_DIM*SUBM_DIM))%SUBM_DIM;
     l = (threads/(SUBM_DIM*SUBM_DIM*SUBM_DIM))%SUBM_DIM;

     //printf("for thread %d, subm index is %d \n", f,
feature_num*SUBM_DIM*SUBM_DIM*vec_size+ POS(k, l, SUBM_DIM, VEC_SIZE));

     mahadist(targetF + POS(j, i, SUBM_DIM, VEC_SIZE),
                subm + feature_num*SUBM_DIM*SUBM_DIM*VEC_SIZE+ POS(k, l,
SUBM_DIM, VEC_SIZE),
                precision_matrix,
                VEC_SIZE,
                (result+f));
     //printf("result for maha call thdidx = %d targetF %d and subm %d is: %f \n", f,
POS(j, i, SUBM_DIM, VEC_SIZE), feature_num*SUBM_DIM*SUBM_DIM*VEC_SIZE+
POS(k, l, SUBM_DIM, VEC_SIZE), *(result+f));
        //printf("inputs were:\ntargetF + POS(i, j, SUBM_DIM, vec_size) = %f\nsubm
+ feature_num*SUBM_DIM*SUBM_DIM*vec_size+ POS(k, l, SUBM_DIM,
vec_size) %f\nvec_size %f\n", (float)*(targetF + POS(i, j, SUBM_DIM,
VEC_SIZE)),(float)*(subm + feature_num*SUBM_DIM*SUBM_DIM*VEC_SIZE+ POS(k,
l, SUBM_DIM, VEC_SIZE)),(float)VEC_SIZE );

  }
```

```
  __syncthreads();
  //if(threadIdx.x == 1)
        //printf("features_blk is %d and threadIdx is %d \n", features_blk, f);
  if(f < features_blk)
  {
        //printf("calculating temp sum for kernel %d, number %d \n", kernelnum,
threadIdx.x);
        temp_sum = 0;
        for(i = 0; i < SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM; i++)
           temp_sum += result[f*SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM + i];
        //printf("The temp sum for f = %d in kernel %d is: %f \n", f, kernelnum,
temp_sum);
        *(out_data + jay*features_blk*NUM_KERNELS + kay*features_blk + f) =
temp_sum;
        //printf("thread %d tempsum is %f \n", f, temp_sum);
  }
        //printf("value written for jay %d and kay %d: %f \n", jay, kay,  *(out_data +
jay*features_blk*NUM_KERNELS + kay*features_blk + f));
     //printf("successfully output data \n");

}
```

```
/*************************************************************************************
************************************/
```

```
// compares a single feature between the target and a predefined number of entries in the database

__global__ void awesomecakes(float * in_data, float * out_data, float * targetF, int jay, int kay, int features_blk, int matrixdim, int vec_size, float * precision_matrix, int kernelnum, int * jj, int * kk)

{

   //printf("kernel number is %d, jvar is %d, kvar is %d \n", kernelnum, jay, kay);

   jay = *jj;

   kay = *kk;

   //printf("Attempt 2 kernel number is %d, jvar is %d, kvar is %d \n", kernelnum, jay, kay);




         // in_data is in_kernel[k]

         // out_data is out_d[i]

   // jay*5 + kay forms the index of the first entry being processed

   // for the index in out_device, compute the distance between the two features in in_device and the target (in constant memory)

   // targetFeature holds the target's feature information



         //int i;

   // allocate an array in shared memory to buffer the output:

                  // WHY do we need the output in shared memory? seems like an extra memory write for nothing since there's not multiple accesses...

   /*__shared__ float outbuf[features_blk];

   // initialize this buffer to 0


   for( i=0; i < features_blk; i++)
```

```
    outbuf[features_blk] = 0;
  */


  //allocate shared memory to buffer the data from the database entries' features.
PARALLELIZE THIS
  /* int data_size;
  data_size =features_blk * vec_size * matrixdim * matrixdim;
  __shared__ float data[FEAT_BLK * VEC_SIZE * DIM_FEAT * DIM_FEAT];
  for( i=0; i< data_size; i++)
    data[i] = *(in_data + i);



  __syncthreads();
  */
  //allocate shared memory for submatricies of target and entries
  __shared__ float subm[((FEAT_BLK * VEC_SIZE * DIM_FEAT *
DIM_FEAT)/(SUBM_DIM*SUBM_DIM))]; //need to include in memory block
allocation calculations in main
  //__shared__ float tsubm[SUBM_DIM*SUBM_DIM*vec_size];
  //int penis;
  /*if(kernelnum == 1)
  {
      for( penis = 0; penis< VEC_SIZE*VEC_SIZE; penis++)
      {
        printf("precmatrix of %d is: %f \n", penis, *(precision_matrix + penis));
      }
  }*/
  // iterate over the database entries, create submatricies for each, and run
calculations with these submatricies
```

```
int f;
for(f = 0; f < features_blk; f++)
{
   //for each entry,

   // form the submatricies
   //form division index for creating sub-matrix vector averages
   int div_idx = matrixdim/SUBM_DIM;

   int i, j, k, l, m;
   float temp_sum;
   //printf("entered f loop \n");
   // average the sub-matricies
   for(i =0; i < SUBM_DIM; i++)
      for(j =0; j < SUBM_DIM; j++)
         for( k = 0; k< vec_size; k++)
         {
            temp_sum = 0;
            for(l = 0; l < div_idx; l++)
               for(m = 0; m < div_idx; m++)
                  temp_sum += *(in_data + f*vec_size*matrixdim*matrixdim + POS(l,m,
matrixdim, vec_size) + POS(i,j, SUBM_DIM, vec_size * div_idx) + k);
                  //temp_sum += *(db_entry.data +
POS(l,m,db_entry.dim_data,db_entry.vec_size) + POS(i,j,SUBM_DIM,
db_entry.vec_size * db_div) + j*db_rem*db_entry.vec_size + k);

            //*(db_vectors + POS(i,j,SUBM_DIM, dim_vec) + k) =
temp_sum/(db_div*db_div);
```

```
            subm[f*SUBM_DIM*SUBM_DIM*vec_size + POS(i,j, SUBM_DIM, vec_size) +
k] = temp_sum/(div_idx*div_idx);
            //maxj*j + k*features_blk + r
            //maxj = ceil((float)size_db/(float)(NUM_KERNELS*features_blk));
            //*(out_data + jay* + kay +f) =
        }
    //printf("created submatricies \n");
    // call the mahalanobis distance on each
    //float min_dist = FLT_MAX;
    float temp[1];

    temp[0] = 0;
    //for( i = 0; i< VEC_SIZE*VEC_SIZE; i++)
      //{
        //*(diff + i) = *(a + i) - *(b + i);
       //   printf("precmatrix of %d is: %f \n", i, *(precision_matrix + i));
      //       }

    for(i = 0; i <SUBM_DIM; i++)
      for(j = 0; j <SUBM_DIM; j++)
        for(k = 0; k<SUBM_DIM; k++)
          for(l = 0; l<SUBM_DIM; l++)
          {
            mahadist(targetF + POS(i, j, SUBM_DIM, vec_size),
                    subm + f*SUBM_DIM*SUBM_DIM*vec_size+ POS(k, l,
SUBM_DIM, vec_size),
                    precision_matrix,
                    vec_size,
```

```
                        temp);
                //printf("t_vectors matrix: \n");
                //gpuprintmatrix(targetFeature + POS(j, i, SUBM_DIM, vec_size), 8);
                //printf("db_vectors matrix: \n");
                //gpuprintmatrix(subm, 8);
                //if(temp[0] < min_dist)
                   //min_dist = temp[0];
            }
    //printf("calculated Mahalanobis distances \n");
    //printf("Minimum distance: %.04f \n", min_dist);
    // store temp to the correct spot in the output
    //float bob = 1;
    //printf("temp value to be written: %f \n", temp[0]);
    //*(out_data + 1) = 1;
    *(out_data + jay*features_blk*NUM_KERNELS + kay*features_blk + f) = temp[0];
    //printf("value written for jay %d and kay %d: %f \n", jay, kay,  *(out_data +
jay*features_blk*NUM_KERNELS + kay*features_blk + f));
    //printf("successfully output data \n");
    //*(out_data + 1) = 1;
    //temp[0];
  }
  //cudaThreadSynchronize();
}


/*********************************************************************************
************************************/


static size_t allocSize = 0;
```

```
//#define myCUDAMalloc(to, size) allocSize += size; printf("Allocating %zd
at %d\n", allocSize, __LINE__);  do{ cudaError_t err = cudaMalloc(to, size); if (err !=
cudaSuccess) { printf("cuda return for %d: %s \n", __LINE__,
cudaGetErrorString(err)); }} while(0)


//#define myCUDAMemcpyAsync(to, from, size, param, stream) allocSize += size;
/*printf("AsyncAllocating %zd at %d\n", allocSize, __LINE__);*/ /*do{ cudaError_t
cpyerr = cudaMemcpy(to, from, size, param); if (cpyerr != cudaSuccess)
{ printf("cuda cpy return for %d: %s \n", __LINE__, cudaGetErrorString(cpyerr)); }}
while(0)*/




int main (int argc, char *argv[])

{

        cudaEvent_t start, stop;

        cudaEventCreate(&start);

        cudaEventCreate(&stop);

        cudaEventRecord(start, 0);



        cudaError_t err;

    feature_offset[0] = 96; //right cheek offset from beginning of entry in bytes

    feature_offset[1] = 112; //left cheek offset " "

    feature_offset[2] = 80; //lips offset " "

    feature_offset[3] = 128; //forehead offset " "

    feature_offset[4] = 144; //hair offset " "
```

```
  //check if we shoul be generating precision matricies from the database instead of runnning the comparison
  if(argc == 4 && (strcmp(argv[1], "gen_prec")==0))
  {
     FILE * d_b = fopen(argv[2], "r");
     FILE * prec_mats = fopen(argv[3], "w");
     if(d_b == NULL || prec_mats == NULL)
     {
        printf("Failed to open data files to generate precision matrix, exiting program.\n");
        return -1;
     }

     if(gen_prec_matricies(d_b, prec_mats) == -1)
     {
        printf("Failed to generate precision matrix file from the given data, program exiting.\n");
        return -1;
     }

     printf("Precision matrix file generated successfully : %s\n", argv[3]);
     return 0;
  }

  //check to make sure we have the right number of args to proceed
  if(argc != 4)
  {
     printf("Please enther the proper number of arguments, program exiting.\n");
```

```
   return -1;
}

//attempt to open database and target information files
FILE * data_base = fopen(argv[3], "r");
FILE * target_dat = fopen(argv[1], "r");
FILE * prec_matricies = fopen(argv[2], "r");

//check to see if the opening files was a success
if(target_dat == NULL || data_base == NULL || prec_matricies == NULL)
{
   printf("Failed to open data files to run comparison, exiting program.\n");
   return -1;
}
printf("Successfully opened files. \n");
//attempt to read in data on our target
entry_t target;
if(read_next_entry((void *)&target, target_dat) == -1)
{
   printf("Target data is corrupted or non-existent, program exiting.\n");
   return -1;
}

//load precision matrix data into memory

printf("Loading precision matrix data into memory. \n");
int dim_rc, dim_lc, dim_lips, dim_fh, dim_hair;
```

```
float * prec_rc_h, * prec_lc_h, * prec_lips_h, * prec_fh_h, * prec_hair_h;

//right cheek precision matrix
if(read_dim((void *)&dim_rc, prec_matricies) == -1)
{
   printf("Unable to read right cheek precision matrix, program exiting.\n");
   return -1;
}

prec_rc_h = (float *)malloc(dim_rc*dim_rc*sizeof(float));



if(read_matrix((void *)prec_rc_h, dim_rc, prec_matricies) == -1)
{
   printf("Unable to read right cheek precision matrix, program exiting.\n");
   return -1;
}
//printf("right cheek allocated. \n");

//left cheek precision matrix
if(read_dim((void *)&dim_lc, prec_matricies) == -1)
{
   printf("Unable to read left cheek precision matrix, program exiting.\n");
   return -1;
}

prec_lc_h = (float *)malloc(dim_lc*dim_lc*sizeof(float));
```

```
  if(read_matrix((void *)prec_lc_h, dim_lc, prec_matricies) == -1)
  {
     printf("Unable to read the left cheek precision matrix, program exiting.\n");
     return -1;
  }
  //printf("left cheek allocated. \n");



  //lips precision matrix
  if(read_dim((void *)&dim_lips, prec_matricies) == -1)
  {
     printf("Unable to read lips precision matrix, program exiting.\n");
     return -1;
  }


  //printf("DIM LIPS
= %i**********************************************************************\n",
dim_lips);


  prec_lips_h = (float *)malloc(dim_lips*dim_lips*sizeof(float));


  if(read_matrix((void *)prec_lips_h, dim_lips, prec_matricies) == -1)
  {
     printf("Unable to read the lips precision matrix, program exiting.\n");
     return -1;
  }
```

```
  //printf("lips allocated************************************ 1st pos = %f. \n",
*prec_lips_h);



  //forehead precision matrix
  if(read_dim((void *)&dim_fh, prec_matricies) == -1)
  {
     printf("Unable to read forehead precision matrix, program exiting.\n");
     return -1;
  }

  prec_fh_h = (float *)malloc(dim_fh*dim_fh*sizeof(float));

  if(read_matrix((void *)prec_fh_h, dim_fh, prec_matricies) == -1)
  {
     printf("Unable to read the forehead precision matrix, program exiting.\n");
     return -1;
  }
  //printf("forehead allocated. \n");



  //hair precision matrix
  if(read_dim((void *)&dim_hair, prec_matricies) == -1)
  {
     printf("Unable to read hair precision matrix, program exiting.\n");
     return -1;
  }
```

```
  prec_hair_h = (float *)malloc(dim_hair*dim_hair*sizeof(float));


  if(read_matrix((void *)prec_hair_h, dim_hair, prec_matricies) == -1)
  {
     printf("Unable to read the hair precision matrix, program exiting.\n");
     return -1;
  }
  //printf("hair allocated. \n");



  printf("Precision matricies read and loaded into memory. \n");
  //write precision matricies to GPU Constant memory


  /*if(cudaMemcpyToSymbol((const char *)prec_rc_c, (void*)prec_rc_h,
sizeof(float)*dim_rc*dim_rc) == cudaErrorInvalidSymbol)
          printf("failed to copy right cheek precision matrix to symbol \n");
  if(cudaMemcpyToSymbol((char *)prec_lc_c, (void*)prec_lc_h,
sizeof(float)*dim_lc*dim_lc) != cudaSuccess)
          printf("failed to copy left cheek precision matrix to symbol \n");
  if(cudaMemcpyToSymbol((char *)prec_lips_c, (void*)prec_lips_h,
sizeof(float)*dim_lips*dim_lips) != cudaSuccess)
          printf("failed to copy lips precision matrix to symbol \n");
  if(cudaMemcpyToSymbol((char *)prec_fh_c, (void*)prec_fh_h,
sizeof(float)*dim_fh*dim_fh) != cudaSuccess)
          printf("failed to copy forehead precision matrix to symbol \n");
  if(cudaMemcpyToSymbol((char *)prec_hair_c, (void*)prec_hair_h,
sizeof(float)*dim_hair*dim_hair) != cudaSuccess)
          printf("failed to copy hair precision matrix to symbol \n");
```

```
  */


  float * prec_rc_d, * prec_lc_d, * prec_lips_d, * prec_fh_d, * prec_hair_d;
  if(cudaMalloc((void**) &prec_rc_d, sizeof(float)*dim_rc*dim_rc) != cudaSuccess)
  {
          printf("Failed to Malloc prec_rc_d \n");
          return -1;
  }
  if(cudaMalloc((void**) &prec_lc_d, sizeof(float)*dim_lc*dim_lc) != cudaSuccess)
  {
          printf("Failed to Malloc prec_lc_d \n");
          return -1;
  }
  if(cudaMalloc((void**) &prec_lips_d, sizeof(float)*dim_lips*dim_lips) !=
cudaSuccess)
  {
          printf("Failed to Malloc prec_lips_d \n");
          return -1;
  }
  if(cudaMalloc((void**) &prec_fh_d, sizeof(float)*dim_fh*dim_fh) != cudaSuccess)
  {
          printf("Failed to Malloc prec_fh_d \n");
     return -1;
  }
  if(cudaMalloc((void**) &prec_hair_d, sizeof(float)*dim_hair*dim_hair) !=
cudaSuccess)
  {
          printf("Failed to Malloc prec_hair_d \n");
```

```
        return -1;
    }


    if(cudaMemcpy(prec_rc_d, prec_rc_h, sizeof(float)*dim_rc*dim_rc,
cudaMemcpyHostToDevice) != cudaSuccess)
    {
        printf("Failed to Load prec_rc_h to device \n");
        return -1;
    }
    if(cudaMemcpy(prec_lc_d, prec_lc_h, sizeof(float)*dim_lc*dim_lc,
cudaMemcpyHostToDevice) != cudaSuccess)
    {
        printf("Failed to Load prec_lc_h to device \n");
        return -1;
    }
    if(cudaMemcpy(prec_lips_d, prec_lips_h, sizeof(float)*dim_lips*dim_lips,
cudaMemcpyHostToDevice) != cudaSuccess)
    {
        printf("Failed to Load prec_lips_h to device \n");
        return -1;
    }
    if(cudaMemcpy(prec_fh_d, prec_fh_h, sizeof(float)*dim_fh*dim_fh,
cudaMemcpyHostToDevice) != cudaSuccess)
    {
        printf("Failed to Load prec_fh_h to device \n");
        return -1;
    }
    if(cudaMemcpy(prec_hair_d, prec_hair_h, sizeof(float)*dim_hair*dim_hair,
cudaMemcpyHostToDevice) != cudaSuccess)
```

```
{
        printf("Failed to Load prec_hair_h to device \n");
        return -1;
}

/* int wha = 0;
int tf = 0;
for(wha = 0; wha < 30; wha ++)
  for(tf = 0; tf< 30; tf ++)
        printf("x: %d y: %d host rc = %f host lc = %f host lips = %f host fh = %f host
hair = %f\n", tf, wha, *(prec_rc_h+tf + wha * 30),*(prec_lc_h+tf + wha *
30),*(prec_lips_h+tf + wha * 30),*(prec_fh_h+tf + wha * 30), *(prec_hair_h+tf + wha *
30) );
*/
// create an array indexing to the precision matricies
float * prec_m[5];
prec_m[0] = prec_rc_d;
prec_m[1] = prec_lc_d;
prec_m[2] = prec_lips_d;
prec_m[3] = prec_fh_d;
prec_m[4] = prec_hair_d;

//attempt to recover the size fo the database we will iterate over
int size_db;

if(read_num_entries((void*)&size_db, data_base) == -1)
{
   printf("Failed to read db, exiting program.\n");
```

```
    return -1;
  }


  //attempt to load the database of entries
  entry_t db[size_db];
  //db = malloc(size_db * sizeof(entry_t));


  int i, j;
  for( i =0; i < size_db; i++)
  {
    if(read_next_entry((void *)&(db[i]), data_base) == -1)
    {
      printf("Database corrupted, program exiting.\n");
      return -1;
    }
  }
  printf("database entries loaded. \n");
  //preform some memory/threading calculations. A possible area for further optimization is to make the calculations specific to each feature since they will all run on separate kernels. In the case of our data , the VEC_SIZE and DIM_FEAT don't vary, but the algoithm can handle variations as long as they are the same for all of a given feature type.
  int features_blk_thds = TGT_THDS_PER_BLK/(THD_PER_FEAT_COMP);
  int features_blk_mem = SHAREDMEM_BLK/((sizeof(float)*VEC_SIZE*SUBM_DIM*SUBM_DIM) + SUBM_DIM*SUBM_DIM*SUBM_DIM*SUBM_DIM*sizeof(float));
  int features_blk = (features_blk_thds < features_blk_mem) ? features_blk_thds : features_blk_mem;
  //features_blk = 2;
  int thds_blk = THD_PER_FEAT_COMP*features_blk;
```

```
  printf("thds_blk is %d \n", thds_blk);



  //create streams for CUDA concurrency and other necessary variables

  cudaStream_t stream1, stream2, stream3, stream4, stream5;

  cudaStreamCreate (&stream1);

  cudaStreamCreate (&stream2);

  cudaStreamCreate (&stream3);

  cudaStreamCreate (&stream4);

  cudaStreamCreate (&stream5);

  cudaStream_t stream[5]; stream[0] = stream1; stream[1] = stream2; stream[2] =
stream3; stream[3] = stream4; stream[4] = stream5;



  printf("threads created. \n");



  //allocate memory on device to be used for computations

  //We need an array holding all the data for the number of features we will be
comparing per kernel for each feature (using pinned page locked memory on host)
This buffer will allow us to use asynchronous memory coppies

  //This part of the code will assume that all feature types have the same vector size
and feature dimensions for better memory allocation efficency and parallelization

  float * in_buf1_d, * in_buf2_d, * in_buf3_d, * in_buf4_d, * in_buf5_d, * in_buf1_h, *
in_buf2_h, * in_buf3_h, * in_buf4_h, * in_buf5_h;

  /*in_buf1_h = (float*)malloc(features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)));

  in_buf2_h = (float*)malloc(features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)));

  in_buf3_h = (float*)malloc(features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)));
```

```
  in_buf4_h = (float*)malloc(features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)));

  in_buf5_h = (float*)malloc(features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)));##*/




  if(cudaMallocHost((void **) &in_buf1_h, features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT))) != cudaSuccess)

  {

     printf("Error allocating memory for in_buf1, program exiting.\n");

     return -1;

  }


  if(cudaMalloc((void **) &in_buf1_d, features_blk *
(sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)) != cudaSuccess)

  {

            printf("Failed to allocate in_buf1_d to device \n");

            return -1;

  }

  /* //to recover host pinned memory pointer on device use this

  if(cudaHostGetDevicePointer((void **) in_buf1_d, (void*) in_buf1_h, 0) !=
cudaSuccess)

  {

     printf("Error mapping memory for in_buf1, program exiting.\n");

     return -1;

  }

  */
```

```
  if(cudaMallocHost((void **) &in_buf2_h, features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT) + sizeof(feature_t))) !=
cudaSuccess)

  {

     printf("Error allocating memory for in_buf2, program exiting.\n");

     return -1;

  }


  if(cudaMalloc((void **) &in_buf2_d, features_blk *
(sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)) != cudaSuccess)

  {

           printf("Failed to allocate in_buf2_d to device \n");

           return -1;

  }

  /*if(cudaHostGetDevicePointer((void **) in_buf2_d, (void*) in_buf2_h, 0) !=
cudaSuccess)

  {

     printf("Error mapping memory for in_buf2, program exiting.\n");

     return -1;

  }

  */


  if(cudaMallocHost((void **) &in_buf3_h, features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT) + sizeof(feature_t))) !=
cudaSuccess)

  {

     printf("Error allocating memory for in_buf3, program exiting.\n");

     return -1;

  }
```

```
  if(cudaMalloc((void **) &in_buf3_d, features_blk *
(sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)) != cudaSuccess)

  {

          printf("Failed to allocate in_buf3_d to device \n");

          return -1;

  }


  /*

  if(cudaHostGetDevicePointer((void **) in_buf3_d, (void*) in_buf3_h, 0) !=
cudaSuccess)

  {

     printf("Error mapping memory for in_buf3, program exiting.\n");

     return -1;

  }

  */


  if(cudaMallocHost((void **) &in_buf4_h, features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT) + sizeof(feature_t))) !=
cudaSuccess)

  {

     printf("Error allocating memory for in_buf4, program exiting.\n");

     return -1;

  }


  if(cudaMalloc((void **) &in_buf4_d, features_blk *
(sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)) != cudaSuccess)

  {

          printf("Failed to allocate in_buf4_d to device \n");
```

```
        return -1;
  }

  /*
  if(cudaHostGetDevicePointer((void **) in_buf4_d, (void*) in_buf4_h, 0) !=
cudaSuccess)
  {
     printf("Error mapping memory for in_buf4, program exiting.\n");
     return -1;
  }
  */

  if(cudaMallocHost((void **) &in_buf5_h, features_blk *
((sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT) + sizeof(feature_t))) !=
cudaSuccess)
  {
     printf("Error allocating memory for in_buf5, program exiting.\n");
     return -1;
  }

  if(cudaMalloc((void **) &in_buf5_d, features_blk *
(sizeof(float)*VEC_SIZE*DIM_FEAT*DIM_FEAT)) != cudaSuccess)
  {
        printf("Failed to allocate in_buf5_d to device \n");
        return -1;
  }

  /*
```

```
  if(cudaHostGetDevicePointer((void **) in_buf5_d, (void*) in_buf5_h, 0) !=
cudaSuccess)
  {
     printf("Error mapping memory for in_buf5, program exiting.\n");
     return -1;
  }
  */


  //create a pointer array for iterative access to buffers
  /*
  float * in_device[5];
  in_device[0] = in_buf1_d;
  in_device[1] = in_buf2_d;
  in_device[2] = in_buf3_d;
  in_device[3] = in_buf4_d;
  in_device[4] = in_buf5_d;
   */
  //printf("In buffers allocated. \n");


  float * in_device[5]; in_device[0] = in_buf1_d; in_device[1] = in_buf2_d; in_device[2]
= in_buf3_d; in_device[3] = in_buf4_d; in_device[4] = in_buf5_d;
  //We need a return array for each feature with one entry for each db entry (uisng
pinned pagelocked memory here to allow access by host and device simultaneously)
  float * out_rc_d, * out_lc_d, * out_lips_d, * out_fh_d, *out_hair_d, * out_rc_h, * out_lc_h,
* out_lips_h, * out_fh_h, *out_hair_h;


  if(cudaMallocHost((void **) &out_rc_h, sizeof(float) * size_db) != cudaSuccess)
  {
     printf("Error allocating memory for out_rc, program exiting.\n");
```

```
    return -1;
  }

  if(cudaHostGetDevicePointer((void **) &out_rc_d, (void*) out_rc_h, 0) !=
cudaSuccess)
  {
    printf("Error mapping memory for out_rc, program exiting.\n");
    return -1;
  }

  if(cudaMallocHost((void **) &out_lc_h, sizeof(float) * size_db) != cudaSuccess)
  {
    printf("Error allocating memory for out_lc, program exiting.\n");
    return -1;
  }

  if(cudaHostGetDevicePointer((void **) &out_lc_d, (void*) out_lc_h, 0) !=
cudaSuccess)
  {
    printf("Error mapping memory for out_lc, program exiting.\n");
    return -1;
  }

  if(cudaMallocHost((void **) &out_lips_h, sizeof(float) * size_db) != cudaSuccess)
  {
    printf("Error allocating memory for out_lips, program exiting.\n");
    return -1;
  }
```

```
  if(cudaHostGetDevicePointer((void **) &out_lips_d, (void*) out_lips_h, 0) !=
cudaSuccess)
  {
     printf("Error mapping memory for out_lips, program exiting.\n");
     return -1;
  }

  if(cudaMallocHost((void **) &out_fh_h, sizeof(float) * size_db) != cudaSuccess)
  {
     printf("Error allocating memory for out_fh, program exiting.\n");
     return -1;
  }

  if(cudaHostGetDevicePointer((void **) &out_fh_d, (void*) out_fh_h, 0) !=
cudaSuccess)
  {
     printf("Error mapping memory for out_fh, program exiting.\n");
     return -1;
  }

  if(cudaMallocHost((void **) &out_hair_h, sizeof(float) * size_db) != cudaSuccess)
  {
     printf("Error allocating memory for out_hair, program exiting.\n");
     return -1;
  }
```

```
  if(cudaHostGetDevicePointer((void **) &out_hair_d, (void*) out_hair_h, 0) !=
cudaSuccess)

  {

     printf("Error mapping memory for out_hair, program exiting.\n");

     return -1;

  }


  //create array for iterative access to device write out locations

  float *out_d[5];

  out_d[0] = out_rc_d; out_d[1] = out_lc_d; out_d[2] = out_lips_d; out_d[3] = out_fh_d;
out_d[4] = out_hair_d;


  //printf("Write out arrays created, out buffers allocated. \n");


  //allocate memory on host for results of CUDA kernel comparisons

  //already done in previous pinned memory allocations (available on host and on
device) since there is only one write out of the final value, it is ok to just write it
directly to the pinned memory from the device to avoid a memcpy call to bring it
back to the host


  //initialize thread block and kernel grid dimensions

  dim3 dim_grid, dim_block;

  dim_grid.x = 1;

  dim_block.x = thds_blk;

  printf("dim_block is %d \n", dim_block.x);


  int r, l, m, k;

  float temp_sum = 0;
```

```
  //t_div is the sqrt(number of pixels in a sub matrix)
  //t_rem represents the remainder of pivels after division by subm_dim and they are not used
  int t_div;
  //int t_rem;

  //prepare targetFeature submatricies
  float * t_subm = (float*)malloc(SUBM_DIM*SUBM_DIM*VEC_SIZE * features_blk * sizeof(float));

  //printf("entering target feature submatricies generation loop. \n");
  //int8_t * ptr = (((int8_t *)&target) + feature_offset[r]);
  for( r = 0; r < FEAT_COUNT; r++) //iterate over rc, lc, lips, hair, etc.
  {
        //printf("target pointer: %p \n", ((int8_t *)&target));

        //printf("target pointer with offset: %p \n", (((int8_t *)&target) + feature_offset[r]));

        //printf("dim_data is: %x \n",(*((int *)(((int8_t *)&target) + feature_offset[r]))));
        //printf("target file first thing is: %x \n",(*(int *)((int8_t *)&target + 100)));
    t_div = (*((int *)(((int8_t *)&target) + feature_offset[r]+4)))/SUBM_DIM;
    //printf("tdiv is: %d \n", t_div);
    //t_rem = (*((int *)(((int8_t *)&target) + feature_offset[r]+4)))%SUBM_DIM;
    //printf("t_div and r_dem initiated. \n");
    //int g;
    //g = sizeof(float *);
    //printf("size of float*: %d \n", g);
```

```
    for(i = 0; i< SUBM_DIM; i++)
      for(j = 0; j < SUBM_DIM; j++)
        for(k = 0; k < VEC_SIZE; k++)
        {
          temp_sum = 0;
          for(l = 0; l < t_div; l++)
            for(m = 0; m < t_div; m++)
            {
                //printf("everything before .data: %p \n", (*((int *)(((int8_t
*)&target) + feature_offset[r] + 12))));
                //printf("everything before .data (chris code): %p \n",
( (*((feature_t *)(((int8_t *)&target) + feature_offset[r]))).data));
              //printf("temp_sum is: %d \n", temp_sum);
              //printf("dereferenced first address: %x \n", (float *)(*((feature_t
*)(((int8_t *)&target) + feature_offset[r] - 4))).data);
              temp_sum += *((*((feature_t *)(((int8_t *)&target) +
feature_offset[r]))).data
                      + j*VEC_SIZE*SUBM_DIM
                      + i*VEC_SIZE*DIM_FEAT*SUBM_DIM
                      + m*VEC_SIZE
                      + l*VEC_SIZE*DIM_FEAT + k);

              /*+ POS(l, m, SUBM_DIM, VEC_SIZE)
              + POS(i,j, SUBM_DIM, VEC_SIZE * t_div)
              + j*t_rem*VEC_SIZE + k);*/
            }
          //printf("temp_sum is: %f \n", temp_sum);
          //printf("t_div is: %d \n", t_div);
```

```
            *(t_subm + POS(i,j,SUBM_DIM, VEC_SIZE) + k +
r*SUBM_DIM*SUBM_DIM*VEC_SIZE) = temp_sum/(t_div*t_div);
            //printf("loaded temp_sum into memory \n");
          }
  }



  float *targetFeature_d;
  if(cudaMalloc((void**)&targetFeature_d,
sizeof(float)*VEC_SIZE*SUBM_DIM*SUBM_DIM) != cudaSuccess)
  {
          printf("Failed to allocate memory for targetFeature_d \n");
          return -1;
  }


  //printf("entering kernel call loop. \n");
  //begin kernel loop iterating over features and concurrently scheduling kernels


  // allocate memory for the j and k variables on the host and device
  int *jvar, *kvar;
  if(cudaMalloc((void**)&jvar, sizeof(int)) != cudaSuccess)
  {
          printf("Failed to allocate memory for jvar \n");
          return -1;
  }
  if(cudaMalloc((void**)&kvar, sizeof(int)) != cudaSuccess)
  {
     printf("Failed to allocate memory for kvar \n");
```

```
    return -1;
  }



  int finalpos;
  for( i =0; i < FEAT_COUNT; i ++)
  {
          finalpos = 0;
          //printf("entering i loop \n");
     //load in current feature submatrix from target to constant memory


          if(cudaMemcpy(targetFeature_d, (const void*)(((int8_t *)t_subm) +
i*sizeof(float)* SUBM_DIM*SUBM_DIM*VEC_SIZE), sizeof(float) *
SUBM_DIM*SUBM_DIM*VEC_SIZE, cudaMemcpyHostToDevice) != cudaSuccess)
          {
                  printf("Failed to copy the feature to targetFeature_d \n");
                  err = cudaPeekAtLastError();
                  printf("Error message: %s \n", cudaGetErrorString(err));
                  return -1;
          }
          //if(cudaMemcpyToSymbol(targetFeature, (const void*)(((int8_t *)t_subm)
+ i*sizeof(float)* SUBM_DIM*SUBM_DIM*VEC_SIZE), sizeof(float) *
SUBM_DIM*SUBM_DIM*VEC_SIZE) != cudaSuccess)
         //  printf("Failed to copy the target's data to symbol \n");



     //begin looping over databse & loading in features from database in accordance
with size calculations
     //printf("size_db: %d \n features_blk: %d \n", size_db, features_blk);
```

```
int maxj;
maxj = (int)ceil(((float)size_db)/((float)(NUM_KERNELS*features_blk))); // maxj represents the maximum value that j can take.
//printf("maxj for i: %d is %d \n", i, maxj);

int offsetsk[maxj*NUM_KERNELS];
offsetsk[0] = 0;
offsetsk[1] = features_blk;
offsetsk[2] = features_blk*2;
offsetsk[3] = features_blk*3;
offsetsk[4] = features_blk*4;

int offsetsj[maxj];
offsetsj[0] = 0;
offsetsj[1] = NUM_KERNELS*features_blk;

for(j = 0; j< maxj; j++) // j loops through batches of kernels of NUM_KERNEL size
{
        //printf("entering j loop. \n");

    for(k = 0; k < NUM_KERNELS && finalpos == 0; k++) //iterates over the kernels in each batch of NUM_KERNELS size
    {
        //if((5*j+k)>=size_db)
                //break;
        //printf("entering k loop \n");
```

```
        //load database entries to be processed into GPU memory. again here we
are assuming that all of the features have the same size dataset of VEC_SIZE and
DIM_FEAT

        for(r = 0; r < features_blk; r++) // load each feature for the current kernel

        {

                //int *ptr = 0;


         //cudaError_t err = cudaMalloc((void**)&ptr, UINT_MAX);

         //printf("for j: %d, k: %d, and r: %d, the check gives %d \n", j, k, r,
(NUM_KERNELS*features_blk*j + k*features_blk + r));

         if((NUM_KERNELS*features_blk*j + k*features_blk + r) < size_db)

         {

                if( cudaMemcpyAsync((void*)(((int8_t *)in_device[k]) +
(r*(VEC_SIZE*DIM_FEAT*DIM_FEAT*sizeof(float)))),

                                (const void*) (*((feature_t*)(((int8_t *)(db
+NUM_KERNELS*j*features_blk + k*features_blk + r)) + feature_offset[i]))).data,

                                VEC_SIZE*DIM_FEAT*DIM_FEAT*sizeof(float),

                                cudaMemcpyHostToDevice,

                                stream[k]) != cudaSuccess)

                {

                        printf("Failed to copy in_device information to device for
iteration %d in kernel number %d and iteration %d  \n", r, i, k);

                        err = cudaPeekAtLastError();

                        printf("Error message: %s \n", cudaGetErrorString(err));

                        return -1;

                }

                //printf("value of r: %d \n", r);

         }

         else
```

```
        {
                //printf("end of run, r is %d \n", r);
                finalpos = features_blk - r;
                break;
        }
       }

       //printf("starting kernel number %d, features_blk-finalpos is %d, j is %d
and k is %d \n", j*NUM_KERNELS+ k, (features_blk-finalpos), j, k);
       //Run comparsion kernel

       //printf("features_blk = %d\nFEAT_BLK = %d\n", features_blk,
(int)FEAT_BLK);

       parallelawesomecakes<<<dim_grid, dim_block, 0,
stream[k]>>>(in_device[k], (out_d[i]), targetFeature_d, j, k, (features_blk-finalpos),
DIM_FEAT, VEC_SIZE, prec_m[i], j*NUM_KERNELS + k, jvar, kvar);

       /*err = cudaGetErrorString(cudaPeekAtLastError());
       printf("%s\n", err);
       err = cudaGetErrorString(cudaThreadSynchronize());
       printf("%s\n", err);*/

       //Asynchronously copy result data out and new feature data in (to be done
in kernel with memory shared between GPU and host through cudaMallocHost)

       cudaDeviceSynchronize();
```

```
        }
    }
    cudaDeviceSynchronize();
    //int chris;
    //for(chris=0; chris<234; chris++)
        //printf("out_d[%d][%d] is: %f \n", i, chris, *(((float*)out_d[i]) + chris));
}


//for(i=0; i<5; i++)
        //for(j=0; j<234; j++)
            //printf("out_d[%d][%d] is: %f \n", i, j, *(((float*)out_d[i]) + j));
//printf("out_rc_h first number has: %f \n", *(out_rc_h));


//free device memory allocations
cudaFree(in_buf1_d); cudaFree(in_buf2_d); cudaFree(in_buf3_d); cudaFree(in_buf4_d); cudaFree(in_buf5_d);
printf("freed the in buffers \n");


//weight the different feature distances and then sum them to form one distance for each db entry
float total_dist[size_db];
for(i = 0; i<size_db; i++)
{
    total_dist[i] = ((*(out_rc_h + i))*W_RCHEEK + (*(out_lc_h + i))*W_LCHEEK + (*(out_lips_h + i))*W_LIPS + (*(out_fh_h + i))*W_FHEAD + (*(out_hair_h + i))*W_HAIR)/(W_RCHEEK+W_LCHEEK+W_LIPS+W_FHEAD+W_HAIR);
    //printf("the total distance value at %d is %f \n", i, total_dist[i]);
}
```

```
  //set up and use a prefix scan kernel to pick out the closest 5 matches (adapt from prefix scan lab)

    //allocate memory for input and output

    float * totals, * out_final;

    int * in_idx, * out_idx;

    cudaMalloc((void **)&totals, sizeof(float) * size_db);

    cudaMalloc((void **)&out_final, sizeof(float) * ceil(((float)size_db)/2560)*5);

/**********MUST ALLOCATE MEMORY FOR INDEXES AND POPULATE*************************************************/


    int max = 0;

    int out_final_h[5]; //for output to host

    out_final_h[0]= 0; out_final_h[1]= 1; out_final_h[2]= 2; out_final_h[3]= 3; out_final_h[4]= 4;


    for(j = 0; j< 5; j++)

    {

        if(total_dist[out_final_h[j]] > total_dist[out_final_h[max]])

                max = j;

    }


    for( i = 4; i < size_db; i++)

    {

        if(total_dist[i] < total_dist[out_final_h[max]])

        {


                out_final_h[max] = i;

                max = 0;
```

```
                for(j = 1; j< 5; j++)
                {
                        if(total_dist[out_final_h[j]] > total_dist[out_final_h[max]])
                                max = j;
                }
        }

    }

/*****************************************************
    //copy total_dist array to GPU (will need to allocate memory for it)
    cudaMemcpy((void *)totals, (const void *)total_dist, sizeof(float) * size_db,
cudaMemcpyHostToDevice);

    //calculate dim_block/dim_grid
    dim_block.x = BLOCK_SIZE;
    dim_grid.x = ceil(((float)size_db)/(((float)BLOCK_SIZE)*10));
    //^EACH THREAD LOOKS AT 10 ENTRIES AT A TIME AND PICKS THE MIN 5.

    //run prefix scan kernel - just modify the one we wrote for class, will need to
treat entries in groups of ten taking the best 5 instead of groups to avoid races
    // isn't this supposed to be reduction??

    reduction<<<dim_block, dim_grid>>>(out_final, out_idx, totals, in_idx, size_db);

    int recursion = (int)ceil((((float)size_db)/2056*5));

    while(recursion > 5)
```

```
    {
        cudaMemcpy((void*) totals, (const void*)out_final, sizeof(float) * recursion,
cudaMemcpyDeviceToDevice);
        dim_grid.x = ceil(((float)recursion)/(((float)BLOCK_SIZE)*10));

        reduction<<<dim_block, dim_grid>>>(out_final, out_idx, totals, in_idx,
recursion);
        recursion = (int)ceil((((float)recursion)/2056*5));
    }

    //copy back result (top 5 closest matches) to host ( will need to allocate memory
for this as well)
    cudaMemcpy((void *)&out_final_h, (const void *)out_final, sizeof(float) * 5,
cudaMemcpyDeviceToHost);

    //cleanup (free any memory allocated)
    cudaFree(totals); cudaFree(out_final);

  //write out the results to the terminal
  //kernel will need to keep track of index number of the lowest values in total_dist
array so we can recover names from database and also access levels as we print out
**********************************************************************************
**************************/

  printf("************************RESULTS************************\n");
  int min;
  for( i = 0; i < 5; i++)
  {
```

```
    min = 0;
    for(j = 1; j <5; j++)
       if(total_dist[out_final_h[j]] < total_dist[out_final_h[i]])
          min = j;

    printf("%d: %s\ndist: %f\n", i, db[out_final_h[min]].name,
total_dist[out_final_h[min]]);
  }
  printf("************************************************************\n");


  //free all allocated host memory
  free(prec_rc_h); free(prec_lc_h); free(prec_lips_h); free(prec_fh_h);
free(prec_hair_h); free(t_subm);
  cudaFreeHost(in_buf1_h); cudaFreeHost(out_rc_h); cudaFreeHost(in_buf2_h);
cudaFreeHost(out_lc_h);
  cudaFreeHost(in_buf3_h); cudaFreeHost(out_lips_h); cudaFreeHost(in_buf4_h);
cudaFreeHost(out_fh_h);
  cudaFreeHost(in_buf5_h); cudaFreeHost(out_hair_h);

  for( i = 0; i<size_db; i++)
    free_entry(db[i]);
  cudaEventRecord(stop,0);
  cudaEventSynchronize(stop);
  float et;
  cudaEventElapsedTime(&et, start, stop);
  cudaEventDestroy(start);
  cudaEventDestroy(stop);
  //printf("%f \n", et);
```

```
    return 0;
}
```