```
#include <stdio.h>


const int DSIZE = 40960;
const int block_size = 256;
const int grid_size = DSIZE/block_size;


__global__ void vector_swap(float *A, float *B, float *C, int DSIZE) {

    //FIXME:
    // Express the vector index in terms of threads and blocks
    int idx =  threadIdx.x + blockIdx.x * blockDim.x;
    // Swap the vector elements - make sure you are not out of range
    if (idx < DSIZE){
        C[idx] = A[idx];
        A[idx] = B[idx];
        B[idx] = C[idx];
    }
}


int main() {


    float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    int print_num = 10;
    h_A = new float[DSIZE];
    h_B = new float[DSIZE];
    h_C = new float[DSIZE];


    for (int i = 0; i < DSIZE; i++) {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
        h_C[i] = 0;
    }

    printf("Old A = [");
    for (int i = 0; i<print_num; i++){
        printf("%f, ", h_A[i]);
    }
    printf("]\n");

    printf("Old B = [");
    for (int i = 0; i<print_num; i++){
        printf("%f, ", h_B[i]);
    }
    printf("]\n");

    // Allocate memory for host and device pointers
    cudaMalloc(&d_A, DSIZE*sizeof(float));
    cudaMalloc(&d_B, DSIZE*sizeof(float));
    cudaMalloc(&d_C, DSIZE*sizeof(float));

    // Copy from host to device
    cudaMemcpy(d_A, h_A, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, DSIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, DSIZE*sizeof(float), cudaMemcpyHostToDevice);

    // Launch the kernel
    vector_swap<<<grid_size, block_size>>>(d_A, d_B, d_C, DSIZE);

    // Copy back to host
    cudaMemcpy(h_A, d_A, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(h_B, d_B, DSIZE*sizeof(float), cudaMemcpyDeviceToHost);

    // Print and check some elements to make sure swapping was successfull
    printf("New A = [");
    for (int i = 0; i<print_num; i++){
        printf("%f, ", h_A[i]);
    }
    printf("]\n");

    printf("New B = [");
    for (int i = 0; i<print_num; i++){
        printf("%f, ", h_B[i]);
    }
    printf("]\n");
    // Free the memory

    cudaFree(d_A);
    cudaFree(d_B);
```

```
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}
```

```
#include <stdio.h>


const int DSIZE_X = 256;
const int DSIZE_Y = 256;
const int block_size = 32;

__global__ void add_matrix(const float *A, const float *B, float *C, int DSIZE_X, int DSIZE_Y)
{
    //FIXME:
    // Express in terms of threads and blocks
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    // Add the two matrices - make sure you are not out of range
    if (idx <  DSIZE_X && idy < DSIZE_Y )
        C[idy*DSIZE_Y + idx] =  A[idy*DSIZE_Y + idx] + B[idy*DSIZE_Y + idx];
}

int main()
{
    float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    int print_num = 3;
    // Create and allocate memory for host and device pointers
    h_A = new float[DSIZE_X * DSIZE_Y];
    h_B = new float[DSIZE_X * DSIZE_Y];
    h_C = new float[DSIZE_X * DSIZE_Y];

    cudaMalloc(&d_A, DSIZE_X * DSIZE_Y*sizeof(float));
    cudaMalloc(&d_B, DSIZE_X * DSIZE_Y*sizeof(float));
    cudaMalloc(&d_C, DSIZE_X * DSIZE_Y*sizeof(float));

    // Fill in the matrices
    // FIXME
    for (int i = 0; i < DSIZE_X; i++) {
        for (int j = 0; j < DSIZE_Y; j++) {
            h_A[i*DSIZE_X + j] = rand()/(float)RAND_MAX;
            h_B[i*DSIZE_X + j] = rand()/(float)RAND_MAX;
            h_C[i*DSIZE_X + j] = 0;
        }
    }

    // Copy from host to device
     // Copy from host to device
    cudaMemcpy(d_A, h_A, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyHostToDevice);

    // Launch the kernel
    // dim3 is a built in CUDA type that allows you to define the block
    // size and grid size in more than 1 dimentions
    // Syntax : dim3(Nx,Ny,Nz)
    dim3 blockSize(block_size, block_size);
    dim3 gridSize(DSIZE_X/block_size, DSIZE_Y/block_size);

    add_matrix<<<gridSize, blockSize>>>(d_A, d_B, d_C, DSIZE_X, DSIZE_Y);

    // Copy back to host
    cudaMemcpy(h_C, d_C, DSIZE_X * DSIZE_Y*sizeof(float), cudaMemcpyDeviceToHost);
    // Print and check some elements to make the addition was succesfull
    printf("A = [");
    for (int i = 0; i < print_num; i++) {
        printf("[ ");
        for (int j = 0; j < print_num; j++) {
            printf("%f, ", h_A[DSIZE_Y*j + i]);
        }
        printf("]\n");
    }
    printf("]\n");

    printf("B = [");
    for (int i = 0; i < print_num; i++) {
        printf("[ ");
        for (int j = 0; j < print_num; j++) {
            printf("%f, ", h_B[DSIZE_Y*j + i]);
        }
        printf("]\n");
    }
    printf("]\n");

    printf("A+B = [");
    for (int i = 0; i < print_num; i++) {
        printf("[ ");
        for (int j = 0; j < print_num; j++) {
```

```
            printf("%f, ", h_C[DSIZE_Y*j + i]);
        }
        printf("]\n");
    }
    printf("]\n");
    // Free the memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}
```

```
#include <stdio.h>
#include <time.h>

const int DSIZE = 256;
const float A_val = 3.0f;
const float B_val = 2.0f;

// error checking macro
#define cudaCheckErrors(msg)                                       \
   do {                                                            \
       cudaError_t __err = cudaGetLastError();                     \
       if (__err != cudaSuccess) {                                 \
           fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n",     \
                   msg, cudaGetErrorString(__err),                 \
                   __FILE__, __LINE__);                            \
           fprintf(stderr, "*** FAILED - ABORTING\n");             \
           exit(1);                                                \
       }                                                           \
   } while (0)

// Square matrix multiplication on CPU : C = A * B
void matrix_mul_cpu(const float *A, const float *B, float *C, int size) {
  //FIXME:
  for (int i=0; i<size; i++){
    for (int j=0; j<size; j++){
        C[i*size+j] = 0.0;
        for (int k=0; k<size; k++){
            C[i*size+j] += A[i*size+k]*B[k*size+j];
        }
    }
  }
}

// Square matrix multiplication on GPU : C = A * B
__global__ void matrix_mul_gpu(const float *A, const float *B, float *C, int size) {

    //FIXME:
    // create thread x index
    // create thread y index
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    // Make sure we are not out of range
    if ((idx < size) && (idy < size)) {
        float temp = 0;
        for (int i = 0; i < size; i++){
            //FIXME : Add dot product of row and column
            temp += A[idx*size+i]*B[i*size+idy];
        }
        C[idx*size+idy] = temp;
    }
}

int main() {

    float *h_A, *h_B, *h_C, *h_Ccpu, *d_A, *d_B, *d_C;
    int print_num = 3;
    // These are used for timing
    clock_t t0, t1, t2, t3;
    double t1sum=0.0;
    double t2sum=0.0;
    double t3sum=0.0;

    // start timing
    t0 = clock();

    // N*N matrices defined in 1 dimention
    // If you prefer to do this in 2-dimentions cupdate accordingly
    h_A = new float[DSIZE*DSIZE];
    h_B = new float[DSIZE*DSIZE];
    h_C = new float[DSIZE*DSIZE];
    h_Ccpu = new float[DSIZE*DSIZE];
    for (int i = 0; i < DSIZE*DSIZE; i++){
        h_A[i] = A_val;
        h_B[i] = B_val;
        h_C[i] = 0;
        h_Ccpu[i] = 0;
    }

    // Initialization timing
    t1 = clock();
    t1sum = ((double)(t1-t0))/CLOCKS_PER_SEC;
    printf("Init took %f seconds.  Begin compute\n", t1sum);

```

```
    // Allocate device memory and copy input data from host to device
    cudaMalloc(&d_A, DSIZE*DSIZE*sizeof(float));
    cudaMalloc(&d_B, DSIZE*DSIZE*sizeof(float));
    cudaMalloc(&d_C, DSIZE*DSIZE*sizeof(float));
    cudaCheckErrors("Allocaiton");
    //FIXME:Add all other allocations and copies from host to device
    cudaMemcpy(d_A, h_A, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, DSIZE*DSIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaCheckErrors("Memory copy Host->Device");
    // Launch kernel
    // Specify the block and grid dimentions
    dim3 block(32,32);  //FIXME
    dim3 grid(DSIZE/32,DSIZE/32); //FIXME
    matrix_mul_gpu<<<grid, block>>>(d_A, d_B, d_C, DSIZE);
    cudaCheckErrors("Kernel Launch");
    // Copy results back to host
    cudaMemcpy(h_C, d_C, DSIZE*DSIZE*sizeof(float), cudaMemcpyDeviceToHost);
    cudaCheckErrors("Memory copy Device->Host");
    // GPU timing
    t2 = clock();
    t2sum = ((double)(t2-t1))/CLOCKS_PER_SEC;
    printf ("Done. GPU Compute took %f seconds\n", t2sum);

    // FIXME
    // Excecute and time the cpu matrix multiplication function
    matrix_mul_cpu(h_A, h_B, h_Ccpu, DSIZE);

    // CPU timing
    t3 = clock();
    t3sum = ((double)(t3-t2))/CLOCKS_PER_SEC;
    printf ("Done. CPU Compute took %f seconds\n", t3sum);


    printf("C_GPU = [");
    for (int i = 0; i < print_num; i++) {
        printf("[ ");
        for (int j = 0; j < print_num; j++) {
            printf("%f, ", h_C[DSIZE*j + i]);
        }
        printf("]\n");
    }
    printf("]\n");

    printf("C_CPU = [");
    for (int i = 0; i < print_num; i++) {
        printf("[ ");
        for (int j = 0; j < print_num; j++) {
            printf("%f, ", h_Ccpu[DSIZE*j + i]);
        }
        printf("]\n");
    }
    printf("]\n");

    // FIXME
    // Free memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);
    free(h_Ccpu);

    return 0;

}

// for DSIZE = 256
// Done. GPU Compute took 0.326956 seconds
// Done. CPU Compute took 0.117284 seconds

// for DSIZE = 512
// Done. GPU Compute took 0.348977 seconds
// Done. CPU Compute took 1.072144 seconds
```