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


#define BLOCK_SIZE 32

const int DSIZE = 256;
const int a = 1;
const int b = 1;

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


// CUDA kernel that runs on the GPU
__global__ void dot_product(const int *A, const int *B, int *C, int N) {

        // FIXME
        // Use atomicAdd
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        if (idx < N) {
        atomicAdd(C, A[idx]*B[idx]);
        }
}

int main() {

        // Create the device and host pointers
        int *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;

        // Fill in the host pointers
        h_A = new int[DSIZE];
        h_B = new int[DSIZE];
        h_C = new int;
        for (int i = 0; i < DSIZE; i++){
                h_A[i] = a;
                h_B[i] = b;
        }

        *h_C = 0;


        // Allocate device memory
        cudaMalloc(&d_A, DSIZE*sizeof(int));
        cudaMalloc(&d_B, DSIZE*sizeof(int));
        cudaMalloc(&d_C, sizeof(int));
        // Check memory allocation for errors
        cudaCheckErrors();
        // Copy the matrices on GPU
        cudaMemcpy(d_A, h_A, DSIZE*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, DSIZE*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_C, h_C, sizeof(int), cudaMemcpyHostToDevice);
        // Check memory copy for errors
        cudaCheckErrors();
        // Define block/grid dimentions and launch kernel
        dot_product<<<DSIZE/BLOCK_SIZE, BLOCK_SIZE>>>(d_A, d_B, d_C, DSIZE);
        // Copy results back to host
        cudaMemcpy(h_C, d_C, sizeof(int), cudaMemcpyDeviceToHost);
    // Check copy for errors
        cudaCheckErrors();
        // Verify result
        printf("A.B = %d\n", *h_C);
        // Free allocated 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 <algorithm>


using namespace std;

#define N 64
#define RADIUS 2
#define BLOCK_SIZE 32


__global__ void stencil_2d(int *in, int *out) {

        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS][BLOCK_SIZE + 2 * RADIUS];
        int gindex_x = threadIdx.x + blockDim.x * blockIdx.x;
        int lindex_x = threadIdx.x + RADIUS;
        int gindex_y = threadIdx.y + blockDim.y * blockIdx.y;
        int lindex_y = threadIdx.y + RADIUS;

        // Read input elements into shared memory
        int size = N + 2 * RADIUS;
        temp[lindex_x][lindex_y] = in[gindex_x*size + gindex_y];
        if (threadIdx.x < RADIUS) {
                temp[lindex_x - RADIUS][lindex_y] = in[(gindex_x - RADIUS)*size + gindex_y];
                temp[lindex_x + BLOCK_SIZE][lindex_y] = in[(gindex_x + BLOCK_SIZE)*size + gindex_y];
        }

        if (threadIdx.y < RADIUS ) {
                temp[lindex_x][lindex_y - RADIUS] = in[(gindex_x)*size + gindex_y - RADIUS];
                temp[lindex_x][lindex_y + BLOCK_SIZE] = in[gindex_x*size + gindex_y + BLOCK_SIZE];
        }
        __syncthreads();
        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS; offset <= RADIUS; offset++){
                result += temp[lindex_x + offset][lindex_y];
                result += temp[lindex_x][lindex_y + offset];
        }
        result -= temp[lindex_x][lindex_y]; // remove ceter overlap counter twice

        // FIXME
        // Store the result
        out[gindex_y+size*gindex_x] = result;
}


void fill_ints(int *x, int n) {
   // Store the result
   // https://en.cppreference.com/w/cpp/algorithm/fill_n
   fill_n(x, n, 1);
}


int main(void) {

        int *in, *out; // host copies of a, b, c
        int *d_in, *d_out; // device copies of a, b, c

        // Alloc space for host copies and setup values
        int size = (N + 2*RADIUS)*(N + 2*RADIUS) * sizeof(int);
        in = (int *)malloc(size); fill_ints(in, (N + 2*RADIUS)*(N + 2*RADIUS));
        out = (int *)malloc(size); fill_ints(out, (N + 2*RADIUS)*(N + 2*RADIUS));

        // Alloc space for device copies
        cudaMalloc(&d_in, size);
        // FIXME
        cudaMalloc(&d_out, size);

        // Copy to device
        cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
        // FIXME
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_2d() kernel on GPU
        int gridSize = (N + BLOCK_SIZE-1)/BLOCK_SIZE;
        dim3 grid(gridSize, gridSize);
        dim3 block(BLOCK_SIZE, BLOCK_SIZE);
        // Launch the kernel
        // Properly set memory address for first element on which the stencil will be applied
        stencil_2d<<<grid,block>>>(d_in + RADIUS*(N + 2*RADIUS) + RADIUS , d_out + RADIUS*(N + 2*RADIUS) +
RADIUS);

        // Copy result back to host
```

```
    83:         // FIXME
    84:         cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    85:
    86:         // Error Checking
    87:         for (int i = 0; i < N + 2 * RADIUS; ++i) {
    88:                 for (int j = 0; j < N + 2 * RADIUS; ++j) {
    89:
    90:                         if (i < RADIUS || i >= N + RADIUS) {
    91:                                 if (out[j+i*(N + 2 * RADIUS)] != 1) {
    92:                                         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", i,j,
out[j+i*(N + 2 * RADIUS)], 1);
    93:                                         return -1;
    94:                                 }
    95:                         }
    96:                         else if (j < RADIUS || j >= N + RADIUS) {
    97:                                 if (out[j+i*(N + 2 * RADIUS)] != 1) {
    98:                                         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", i,j,
out[j+i*(N + 2 * RADIUS)], 1);
    99:                                         return -1;
   100:                                 }
   101:                         }
   102:                         else {
   103:                                 if (out[j+i*(N + 2 * RADIUS)] != 1 + 4 * RADIUS) {
   104:                                         printf("Mismatch at index [%d,%d], was: %d, should be: %d\n", i,j,
out[j+i*(N + 2 * RADIUS)], 1 + 4*RADIUS);
   105:                                         return -1;
   106:                                 }
   107:                         }
   108:                 }
   109:         }
   110:
   111:         // Cleanup
   112:         free(in);
   113:         free(out);
   114:         cudaFree(d_in);
   115:         cudaFree(d_out);
   116:         printf("Success!\n");
   117:
   118:         return 0;
   119: }
   120:
   121:
```