# Versión en GPU

#include <stdio.h>

#include <cuda\_runtime.h>

#include <iostream>

#include <cstdlib>

#include <curand.h>

#include <curand\_kernel.h>

#include <ctime>

#include <fstream>

// Función que inicializa la semilla del generador de números aleatorios de cuda

\_\_global\_\_ void setup\_kernel(curandState \* state, unsigned long seed)

{

int id = threadIdx.x;

curand\_init(seed, id, 0, &state[id]);

}

//Función que crea una jewel utilizando globalState

\_\_device\_\_ float generate(curandState\* globalState, int ind)

{

curandState localState = globalState[ind];

float rndm = curand\_uniform(&localState);

globalState[ind] = localState;

return rndm;

}

// Función que crea una jewel aleatoria en CUDA

\_\_device\_\_ int createJewelCUDA(curandState\* globalState, int ind, int difficulty)

{

switch (difficulty) {

case 1:

{

return (int)1 + generate(globalState, ind) \* 4;

}

case 2: {

return (int)1 + generate(globalState, ind) \* 6;

}

case 3: {

return (int)1 + generate(globalState, ind) \* 8;

}

}

return -1;

}

// Función que rellena inicialmente el tablero

\_\_global\_\_ void initialTablePopulation(float \*table, int difficulty, int width, int height, curandState\* globalState) {

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < height)

{

if (tcolumn < width)

{

table[trow\*width + tcolumn] = createJewelCUDA(globalState, trow \* width + tcolumn, difficulty);

}

}

}

void printTable(float\* table, int width, int height) {

for (int i = height - 1; i >= 0; i--) {

printf("\n");

for (int j = 0; j < width; j++) {

printf("%d ", (int)table[j + i\*width]);

}

}

printf("\n");

}

//Función que, en base a las coordenadas de su hilo de ejecución, determina si la jewel correspondiente ha de ser eliminada y hace descender las filas

//correspondientes en base a ello. Utiliza una copia del tablero para evitar condiciones de carrera.

\_\_global\_\_ void jewelErasingKernel(float\* table\_d, float\* table\_aux\_d, float\* erasedJewels\_d, int difficulty, int width, int height, int end, curandState\* globalState) {

int tx = threadIdx.x;

int ty = threadIdx.y;

if (erasedJewels\_d[0] != erasedJewels\_d[2] && tx >= erasedJewels\_d[0] && tx <= erasedJewels\_d[end - 2] && ty >= erasedJewels\_d[1]) {

if (ty + 1 < height) {

float value = table\_aux\_d[tx + (ty + 1)\*width];

table\_d[tx + (ty)\*(width)] = value;

}

else {

table\_d[tx + ty\*width] = createJewelCUDA(globalState, tx + ty\*width, difficulty);

}

}

else {

if (ty < height && tx == erasedJewels\_d[0] && ty > erasedJewels\_d[1]) {

float value = table\_aux\_d[tx + (ty)\*width];

table\_d[tx + (ty - end / 2)\*(width)] = value;

}

if (ty >= height - end / 2 && ty < height && tx == erasedJewels\_d[0]) {

table\_d[tx + (ty)\*width] = createJewelCUDA(globalState, tx + ty\*width, difficulty);

}

}

}

//Función que prepara y ejecuta la eliminación de jewels en GPU.

void eraseJewels(float\* table, float\* erasedJewels, int difficulty, int width, int height, curandState\* globalState) {

float \*table\_d;

float \*erasedJewels\_d;

float \*table\_aux\_d;

int size = width \* height \* sizeof(float);

int max = 0;

//Determina cuántas jewels se pueden, como mucho, eliminar en una dirección.

if (height >= width) max = height;

else max = width;

//Envía a GPU el tablero y una copia

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

cudaMalloc((void\*\*)&table\_aux\_d, size);

cudaMemcpy(table\_aux\_d, table, size, cudaMemcpyHostToDevice);

//Envía las jewels a eliminar a GPU. Se necesita reservar dos veces la cantidad de memoria ocupada

//por max\*sizeof(float) porque cada jewel se identifica mediante dos cifras.

cudaMalloc((void\*\*)&erasedJewels\_d, 2 \* max \* sizeof(float));

cudaMemcpy(erasedJewels\_d, erasedJewels, 2 \* max \* sizeof(float), cudaMemcpyHostToDevice);

int end = 0;

bool altered = false;

//Calcula cuál es el último valor escrito de entre todas las jewels a eliminar

for (int i = 0; i < max \* 2; i++) {

if (erasedJewels[i] < 0) {

end = i;

altered = true;

break;

}

}

if (!altered) end = max \* 2;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

jewelErasingKernel <<<dimGrid, dimBlock >>> (table\_d, table\_aux\_d, erasedJewels\_d, difficulty, width, height, end, globalState);

//Recuperación del nuevo tablero

cudaMemcpy(table, table\_d, size, cudaMemcpyDeviceToHost);

cudaFree(table\_d);

cudaFree(erasedJewels\_d);

cudaFree(table\_aux\_d);

}

// Almacena en una tablero auxiliar todas las jewels eliminadas al realizar un movimiento dado desde una posición.

// El paralelismo se consigue realizando una exploración hacia la derecha.

\_\_global\_\_ void autoTableAnalysisKernel(float \*table\_d, float \*aux\_d, int difficulty, int width, int height) {

int tx = threadIdx.x;

int ty = threadIdx.y;

int potentialJewelsRight = 0;

//Explorar a la derecha

if ((tx + 2) < width) {

if (((tx + 2) + ty\*width <= height\*width) && table\_d[tx + 2 + ty\*width] == table\_d[tx + ty\*width]) {

int i = 2;

while ((tx + i + ty\*width <= height\*width) && table\_d[tx + i + ty\*width] == table\_d[tx + ty\*width]) {

potentialJewelsRight++;

i++;

}

aux\_d[tx + ty\*width] = potentialJewelsRight + 1;

}

else {

aux\_d[tx + ty\*width] = 1;

}

}

else {

aux\_d[tx + ty\*width] = 1;

}

}

//Análisis del tablero tras un movimiento manual, en base a las nuevas coordenadas de la jewel desplazada

void manualTableAnalysis(int difficulty, float\* table, int width, int height, int x, int y, curandState\* globalState) {

int max = 0;

int size = width\*height;

if (height >= width) max = height;

else max = width;

//Se realiza la eliminación de jewels, registrando su posición (x, y)

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

//Valores inicalizads a -1 por la posibilidad de que no se pueda eliminar nada

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

int potentialJewelsRight = 0;

//Explorar por la izquierda

if ((x - 1 + y\*width >= 0) && table[x - 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x - i + y\*width >= 0) && (x - i >= 0) && table[x - i + y\*width] == table[x + y\*width]) {

potentialJewelsRight++;

i++;

}

}

//Explorar por la derecha

if ((x + 1 + y\*width <= size) && table[x + 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x + i + y\*width <= size) && (x + i < width) && table[x + i + y\*width] == table[x + y\*width]) {

potentialJewelsRight++;

i++;

}

}

//Existe la posibilidad de realizar una eliminación horizontal.

if (1 + potentialJewelsRight + potentialJewelsRight >= 3) {

int stride = 0;

for (int j = potentialJewelsRight; j >= (1); j--) {

erasedJewels[stride] = x - j;

erasedJewels[stride + 1] = y;

stride += 2;

}

erasedJewels[potentialJewelsRight \* 2] = x;

erasedJewels[potentialJewelsRight \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsRight; k++) {

erasedJewels[stride + potentialJewelsRight \* 2] = x + k;

erasedJewels[stride + potentialJewelsRight \* 2 + 1] = y;

stride += 2;

}

}

else { //Análisis en vertical

int potentialJewelsAbove = 0;

int potentialJewelsBelow = 0;

//Explorar por abajo

if ((x + (y - 1)\*width >= 0) && table[x + (y - 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y - i)\*width >= 0) && table[x + (y - i)\*width] == table[x + y\*width]) {

potentialJewelsBelow++;

i++;

}

}

//Explorar por arriba

if ((x + 1 + y\*width <= size) && table[x + (y + 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y + i)\*width <= size) && table[x + (y + i)\*width] == table[x + y\*width]) {

potentialJewelsAbove++;

i++;

}

}

//Existe posibilidad de elimar jewels

if (1 + potentialJewelsBelow + potentialJewelsAbove >= 3) {

int stride = 0;

for (int j = potentialJewelsBelow; j >= (1); j--) {

erasedJewels[stride] = x;

erasedJewels[stride + 1] = y - j;

stride += 2;

}

erasedJewels[potentialJewelsBelow \* 2] = x;

erasedJewels[potentialJewelsBelow \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsAbove; k++) {

erasedJewels[stride + potentialJewelsBelow \* 2] = x;

erasedJewels[stride + potentialJewelsBelow \* 2 + 1] = y + k;

stride += 2;

}

}

}

eraseJewels(table, erasedJewels, difficulty, width, height, globalState);

free(erasedJewels);

}

//Desplaza la jewel seleccionada en la dirección indicada. Esta función asume que el movimiento es posible.

void swapSpots(float\* table, int jewel1X, int jewel1Y, int direction, int width, int height, int selection, int difficulty, curandState\* globalState) {

int jewel2X = jewel1X;

int jewel2Y = jewel1Y;

switch (direction)

{

case 1: //Arriba

{

jewel2Y += 1;

break;

}

case 2: //Abajo

{

jewel2Y -= 1;

break;

}

case 3: //Izquierda

{

jewel2X -= 1;

break;

}

case 4: //Derecha

{

jewel2X += 1;

break;

}

}

int aux1;

aux1 = table[jewel2X + jewel2Y\*width];

table[jewel2X + jewel2Y\*width] = table[jewel1X + jewel1Y\*width];

table[jewel1X + jewel1Y\*width] = aux1;

//Comprueba si el movimiento permite la eliminación de jewels

manualTableAnalysis(difficulty, table, width, height, jewel2X, jewel2Y, globalState);

}

//Busca la mejor opción en GPU y la ejecuta

void automaticTableAnalysis(int difficulty, float\* table, int width, int height, curandState\* globalState) {

float \*table\_d;

float \*aux\_d;

float \*aux;

//Tamaño del tablero en memoria

int size = width \* height \* sizeof(float);

int tam = width \* height;

int max = 0;

if (height >= width) max = height;

else max = width;

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

aux = (float\*)malloc(size);

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

for (int p = 0; p < tam; p++) {

aux[p] = 1;

}

//Copia del tablero a GPU

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

//Variable auxiliar para realizar conteo en GPU

cudaMalloc((void\*\*)&aux\_d, size);

cudaMemcpy(aux\_d, aux, size, cudaMemcpyHostToDevice);

//Configuracion de la ejecución

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

//Arranque del kernel

autoTableAnalysisKernel <<<dimGrid, dimBlock >>> (table\_d, aux\_d, difficulty, width, height);

//Copia del resultado de GPU a host

cudaMemcpy(aux, aux\_d, size, cudaMemcpyDeviceToHost);

int bestX = 0;

int bestY = 0;

int valor\_mejor = 0;

//Se exploran todas las posibilidades en busca del mejor movimiento posible

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

if (aux[x + y\*width] > valor\_mejor) {

valor\_mejor = aux[x + y\*width];

bestX = x;

bestY = y;

}

}

}

//Ejecuta el movimiento en caso de ser posible

if (valor\_mejor >= 3) {

swapSpots(table, bestX, bestY, 4, width, height, 1, difficulty, globalState);

}

free(aux);

free(erasedJewels);

cudaFree(table\_d);

cudaFree(aux\_d);

}

bool preloadGame(int& width, int& height, int& difficulty, char\* row)

{

std::ifstream fwidth("width.txt");

if (!fwidth.is\_open())

{

std::cerr << "ERROR: Archivo de guardado (width.txt) no encontrado." << std::endl;

return false;

}

fwidth >> width;

fwidth.close();

std::ifstream fheight("height.txt");

if (!fheight.is\_open())

{

std::cout << "ERROR: Archivo de guardado (height.txt) no encontrado." << std::endl;

return false;

}

fheight >> height;

fheight.close();

std::ifstream fdifficulty("difficulty.txt");

if (!fdifficulty.is\_open())

{

std::cout << "ERROR: Archivo de guardado (difficulty.txt) no encontrado." << std::endl;

return false;

}

fdifficulty >> difficulty;

fdifficulty.close();

std::ifstream fLoad(row);

if (!fLoad.is\_open())

{

std::cout << "ERROR: Archivo de guardado no encontrado." << std::endl;

return false;

}

fLoad.close();

return true;

}

void loadFile(int width, int height, float\* table, char\* row)

{

int aux;

char\* array = (char\*)malloc(width\*height + 1);

std::ifstream fLoad(row);

fLoad.getline(array, width\*height + 1);

for (int i = 0; i < width\*height; i++)

{

aux = (array[i] - 48);

table[i] = (float)aux;

}

free(array);

fLoad.close();

}

void saveFile(float\* table, int width, int height, int difficulty, char\* row)

{

//Sistema de guardado

std::ofstream rowWidth;

rowWidth.open("width.txt");

rowWidth.clear();

rowWidth << width;

rowWidth.close();

std::ofstream rowHeight;

rowHeight.open("height.txt");

rowHeight.clear();

rowHeight << height;

rowHeight.close();

std::ofstream rowDifficulty;

rowDifficulty.open("difficulty.txt");

rowDifficulty.clear();

rowDifficulty << difficulty;

rowDifficulty.close();

std::ofstream saveF;

saveF.open(row);

saveF.clear();

for (int index = 0; index < width\*height; index++)

{

saveF << table[index];

}

saveF.close();

}

// Función que elimina una fila completa

\_\_global\_\_ void rowBomb(float\* table, int width, int height, int difficulty, int row, curandState\* globalState) {

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if ((trow + row) < height)

{

if (tcolumn < width)

{

if ((trow + row + 1) == height)

{

table[(trow + row)\*width + tcolumn] = createJewelCUDA(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

table[(trow + row)\*width + tcolumn] = table[(trow + row + 1)\*width + tcolumn];

}

}

}

}

// Función que elimina una columna completa

\_\_global\_\_ void columnBomb(float\* table, int width, int height, int difficulty, int column, curandState\* globalState) {

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < height)

{

if ((tcolumn + column) < width)

{

if ((column - tcolumn - 1) < 0)

{

table[(trow\*width) + (column - tcolumn)] = createJewelCUDA(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

table[(trow\*width) + (column - tcolumn)] = table[(trow\*width) + (column - tcolumn - 1)];

}

}

}

}

\_\_global\_\_ void pivotBombGPU(float\* table, int width, int height, int row, int column)

{

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < 3)

{

if (tcolumn < 3)

{

table[(row + 1 - tcolumn)\*width + (column - 1 + trow)] = table[((row + 1) - trow)\*width + ((column + 1) - tcolumn)];

}

}

}

\_\_global\_\_ void pivotBomb(float\* table\_d, int width, int height)

{

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < height && tcolumn < width) {

if (!((trow - 1) < 0 || (trow + 1) >= height || (tcolumn - 1) < 0 || (tcolumn + 1) >= width))

{

if (trow % 3 == 1 && tcolumn % 3 == 1)

{

dim3 dimBlock(3, 3);

dim3 dimGrid(1, 1);

pivotBombGPU <<<dimGrid, dimBlock >>> (table\_d, width, height, trow, tcolumn);

}

}

}

}

int main(int argc, char\*\* argv) {

int width;

int height;

int difficulty;

char mode;

int size;

char saveF[9] = "save.txt";

int selection;

float\* table;

float\* table\_d;

curandState\* devStates;

//Configuración inicial del juego

if (argc == 1)

{

std::cout << "Anchura del tablero: ";

std::cin >> width;

std::cout << "Altura del tablero: ";

std::cin >> height;

std::cout << "Elija dificultad:"<<std::endl<<"\t1.-\tFacil"<<std::endl<<"\t2.-\tMedia"<<std::endl<<"\t3.-\tDificil\n";

std::cin >> difficulty;

std::cout << "Jugar automaticamente?"<<std::endl<<"\t1.-\tSI"<<std::endl<<"\t2.-\tNO"<<std::endl;

std::cin >> selection;

}

else

{

mode = argv[1][1];

difficulty = atoi(argv[2]);

width = atoi(argv[3]);

height = atoi(argv[4]);

switch (mode) {

case 'a': {selection = 1;

break; }

case 'm': {selection = 2;

break; }

default: std::cerr<<"ERROR: Valor de mode inválido. Por favor, inserte 'a' o 'm' como mode de juego en la línea de comandos."<<std::endl;

return -1;

}

}

bool playing = true;

size = width\*height;

//Inicialización del rndm de CUDA

cudaMalloc(&devStates, size \* sizeof(curandState));

setup\_kernel <<< 1, size >>> (devStates, unsigned(time(NULL)));

//Creación de punteros al tablero y la versión en GPU del mismo

table = (float\*)malloc(size \* sizeof(float));

cudaMalloc((void\*\*)&table\_d, size \* sizeof(float));

//Inicialización del tablero

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

initialTablePopulation <<<dimGrid, dimBlock >>>(table\_d, difficulty, width, height, devStates);

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

//Bucle de juego: se muestran el estado actual y las opciones disponibles, y se procesa la selección del usuario.

while (playing) {

printTable(table, width, height);

int jewel1X = 0;

int jewel1Y = 0;

int command = 0;

std::cout << "Acción a realizar:"<<std::endl;

std::cout << "1.-\tIntercambiar Jewels"<<std::endl;

std::cout << "2.-\tGuardar partida"<<std::endl;

std::cout << "3.-\tCargar partida"<<std::endl;

std::cout << "9.-\tUsar una bomba"<<std::endl;

std::cout << "0.-\tSalir"<<std::endl;

std::cout << "Inserte seleccion: ";

std::cin >> command;

switch (command) {

case 0: {

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}

//El jugador desea desplazar una jewel. El juego consulta la jewel a desplazar y la dirección, confirmando que el movimiento

//seleccionado es válido.

case 1: {

if (selection == 2)

{

std::cout << "Posicion de la primera jewel a intercambiar (empiezan en 0)\n";

std::cout << "column: ";

std::cin >> jewel1X;

std::cout << "row: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion erronea.\n");

continue;

}

int direction = 0;

std::cout << "Posicion de la jewel (el primer valor es 0):"<<std::endl;

std::cout << "\tColumna: ";

std::cin >> jewel1X;

std::cout << "\tFila: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion invalida.\n");

continue;

}

std::cout << "Direccion del movimiento:<<"<<std::endl<<"\t1.-\tArriba"<<std::endl<<"\t2.-\tAbajo"<<std::endl<<"\t3.-\tIzquierda"<<std::endl<<"\t4.-\tDerecha"<<std::endl;

std::cin >> direction;

if (direction > 4 && direction > 1) {

printf("Movimiento invalido.\n");

continue;

}

else {

switch (direction)

{

case 1: //Arriba

{

if (jewel1Y == height)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 2: //Abajo

{

if (jewel1Y == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 3: //Izquierda

{

if (jewel1X == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 4: //Derecha

{

if (jewel1X == width - 1)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

}

}

//Intercambio entre las jewels

swapSpots(table, jewel1X, jewel1Y, direction, width, height, selection, difficulty, devStates);

}

else if (selection == 1)

{

//Modo automático activado, se procede a la selección y ejecución de movimiento

automaticTableAnalysis(difficulty, table, width, height, devStates);

}

break;

}

//Opción guardar: el estado de la partida actual se escribe a un fichero externo.

case 2: {

saveFile(table, width, height, difficulty, saveF);

std::cout << "Guardado completado"<<std::endl;

break;

}

//Opción cargar: se lee el estado de la partida guardada y se recrea

case 3: {

//El juego prepara la carga del tablero

int found = preloadGame(width, height, difficulty, saveF);

size = width\*height;

if (found)

{

free(table);

table = (float\*)malloc(size \* sizeof(float));

// Carga efectiva del tablero

loadFile(width, height, table, saveF);

std::cout << "Juego automatico?"<<std::endl<<"\t1.-\tSI"<<std::endl<<"\t2.-\tNO"<<std::endl;

std::cin >> selection;

std::cout << "Se ha cargado el estado del tablero:"<<std::endl;

}

else {

std::cout << "No hay partidas guardadas."<<std::endl;

}

break;

}

case 9: {

int bomb = 0;

int row = 0; int column = 0;

std::cout << "Elija una bomba:";

/\*

Selección del tipo de bomba y ejecución del mismo. Los tipos disponibles depende del nivel de dificultad seleccionado,

de acuerdo a las condiciones establecidas en el documento de la práctica.

\*/

switch (difficulty) {

case 1: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb != 1)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, devStates);

break;

}

case 2: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 2)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, devStates);

break;

}

case 2:

{

std::cout << "Columna: ";

std::cin >> column;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

columnBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, devStates);

break;

}

}

break;

}

case 3: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "3.-\tBomba de rotacion 3x3"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 3)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "X: ";

std::cin >> row;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, devStates);

break;

}

case 2:

{

std::cout << "Y: ";

std::cin >> column;

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

columnBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, devStates);

break;

}

case 3:

{

dim3 dimBlock(width, height);

dim3 dimGrid(1, 1);

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

pivotBomb <<<dimGrid, dimBlock >>>(table\_d, width, height);

break;

}

}

break;

}

}

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

break;

}

}

}

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}

# Versión en GPU usando bloques

#include <stdio.h>

#include <cuda\_runtime.h>

#include <iostream>

#include <cstdlib>

#include <curand.h>

#include <curand\_kernel.h>

#include <ctime>

#include <fstream>

//Compruebas las propiedades de la tarjeta gráfica para generar una tesela adecuada, al tiempo que evalua el tamaño del tablero.

int getTileWidth(int width, int height) {

float minSize = 0;

if (width > height) minSize = width;

else minSize = height;

cudaDeviceProp properties;

cudaGetDeviceProperties(&properties, 0);

int maxThreads = properties.maxThreadsPerBlock;

if (width == height) { //En caso de una matriz cuadrada, queremos tener más de un bloque

if (minSize / 32 > 1 && maxThreads == 1024) { //El bloque solo podrá ser de 32x32 si tiene 1024 hilos o más

return 32;

}

else if (minSize / 16 > 1) {

return 16;

}

else if (minSize / 8 > 1) {

return 8;

}

else if (minSize / 4 > 1) {

return 4;

}

else if (minSize / 2 > 1) {

return 2;

}

}

else {

if (minSize / 32 >= 1 && maxThreads == 1024) {

return 32;

}

else if (minSize / 16 >= 1) {

return 16;

}

else if (minSize / 8 >= 1) {

return 8;

}

else if (minSize / 4 >= 1) {

return 4;

}

else if (minSize / 2 >= 1) {

return 2;

}

}

return -1;

}

// Función que inicializa la semilla del generador de números aleatorios de cuda

\_\_global\_\_ void setup\_kernel(curandState \* state, unsigned long seed)

{

int id = threadIdx.x;

curand\_init(seed, id, 0, &state[id]);

}

//Función que crea una jewel utilizando globalState

\_\_device\_\_ float generate(curandState\* globalState, int ind)

{

curandState localState = globalState[ind];

float rndm = curand\_uniform(&localState);

globalState[ind] = localState;

return rndm;

}

// Función que crea una jewel aleatoria en CUDA

\_\_device\_\_ int createJewel(curandState\* globalState, int ind, int difficulty)

{

switch (difficulty) {

case 1:

{

return (int)1 + generate(globalState, ind) \* 4;

}

case 2: {

return (int)1 + generate(globalState, ind) \* 6;

}

case 3: {

return (int)1 + generate(globalState, ind) \* 8;

}

}

return -1;

}

// Función que rellena inicialmente el tablero

\_\_global\_\_ void rndmTableInit(float \*table, int difficulty, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height)

{

if (tcolumn < width)

{

table[trow\*width + tcolumn] = createJewel(globalState, trow \* width + tcolumn, difficulty);

}

}

}

void printTable(float\* table, int width, int height) {

for (int i = height - 1; i >= 0; i--) {

printf("\n");

for (int j = 0; j < width; j++) {

printf("%d ", (int)table[j + i\*width]);

}

}

printf("\n");

}

//Función que, en base a las coordenadas de su hilo de ejecución, determina si la jewel correspondiente ha de ser eliminada y hace descender las filas

//correspondientes en base a ello. Utiliza una copia del tablero para evitar condiciones de carrera.

\_\_global\_\_ void eraseJewelsKernel(float\* table\_d, float\* auxTable\_d, float\* erasedJewels\_d, int difficulty, int width, int height, int end, int TILE\_WIDTH, curandState\* globalState) {

int tx = threadIdx.x;

int ty = threadIdx.y;

int block\_x = blockIdx.x;

int block\_y = blockIdx.y;

//Posicion real dentro del table

tx += block\_x \* TILE\_WIDTH;

ty += block\_y \* TILE\_WIDTH;

if (erasedJewels\_d[0] != erasedJewels\_d[2] && tx >= erasedJewels\_d[0] && tx <= erasedJewels\_d[end - 2] && ty >= erasedJewels\_d[1]) {

if (ty + 1 < height) {

float value = auxTable\_d[tx + (ty + 1)\*width];

table\_d[tx + (ty)\*(width)] = value;

}

else {

table\_d[tx + ty\*width] = createJewel(globalState, tx + ty\*width, difficulty);

}

}

else {

if (ty < height && tx == erasedJewels\_d[0] && ty > erasedJewels\_d[1]) {

float value = auxTable\_d[tx + (ty)\*width];

table\_d[tx + (ty - end / 2)\*(width)] = value;

}

if (ty >= height - end / 2 && ty < height && tx == erasedJewels\_d[0]) {

table\_d[tx + (ty)\*width] = createJewel(globalState, tx + ty\*width, difficulty);

}

}

}

//Función que prepara y ejecuta la eliminación de jewels en GPU.

void eraseJewels(float\* table, float\* erasedJewels, int difficulty, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

float \*table\_d;

float \*erasedJewels\_d;

float \*auxTable\_d;

int size = width \* height \* sizeof(float);

int max = 0;

//Determina cuántas jewels se pueden, como mucho, eliminar en una dirección.

if (height >= width) max = height;

else max = width;

//Envía a GPU el tablero y una copia

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

cudaMalloc((void\*\*)&auxTable\_d, size);

cudaMemcpy(auxTable\_d, table, size, cudaMemcpyHostToDevice);

//Envía las jewels a eliminar a GPU. Se necesita reservar dos veces la cantidad de memoria ocupada

//por max\*sizeof(float) porque cada jewel se identifica mediante dos cifras.

cudaMalloc((void\*\*)&erasedJewels\_d, 2 \* max \* sizeof(float));

cudaMemcpy(erasedJewels\_d, erasedJewels, 2 \* max \* sizeof(float), cudaMemcpyHostToDevice);

int end = 0;

bool altered = false;

//Calcula cuál es el último valor escrito de entre todas las jewels a eliminar

for (int i = 0; i < max \* 2; i++) {

if (erasedJewels[i] < 0) {

end = i;

altered = true;

break;

}

}

if (!altered) end = max \* 2;

//Calcula las dimensiones en función de TILE\_WIDTH

int wdth = ceil(((double)width) / TILE\_WIDTH);

int hght = ceil(((double)height) / TILE\_WIDTH);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

eraseJewelsKernel <<<dimGrid, dimBlock >>> (table\_d, auxTable\_d, erasedJewels\_d, difficulty, width, height, end, TILE\_WIDTH, globalState);

//Recuperación del nuevo tablero

cudaMemcpy(table, table\_d, size, cudaMemcpyDeviceToHost);

//Libera memoria

cudaFree(table\_d);

cudaFree(erasedJewels\_d);

cudaFree(auxTable\_d);

}

// Almacena en una tablero auxiliar todas las jewels eliminadas al realizar un movimiento dado desde una posición.

// El paralelismo se consigue realizando una exploración hacia la derecha.

\_\_global\_\_ void automaticTableAnalysisKernel(float \*table\_d, float \*aux\_d, int difficulty, int width, int height, int TILE\_WIDTH) {

int tx = threadIdx.x;

int ty = threadIdx.y;

int block\_x = blockIdx.x;

int block\_y = blockIdx.y;

//Cálculo de la posición real dentro del tablero

tx += block\_x \* TILE\_WIDTH;

ty += block\_y \* TILE\_WIDTH;

int potentialJewelsRight = 0;

//Exploración por la derecha

if ((tx + 2) < width) {

if (((tx + 2) + ty\*width <= height\*width) && table\_d[tx + 2 + ty\*width] == table\_d[tx + ty\*width]) {

int i = 2;

while ((tx + i + ty\*width <= height\*width) && table\_d[tx + i + ty\*width] == table\_d[tx + ty\*width]) {

potentialJewelsRight++;

i++;

}

aux\_d[tx + ty\*width] = potentialJewelsRight + 1;

}

else {

aux\_d[tx + ty\*width] = 1;

}

}

else {

aux\_d[tx + ty\*width] = 1;

}

}

//Análisis del tablero tras un movimiento manual, en base a las nuevas coordenadas de la jewel desplazada

void manualTableAnalysis(int difficulty, float\* table, int width, int height, int x, int y, int TILE\_WIDTH, curandState\* globalState) {

int max = 0;

int size = width\*height;

if (height >= width) max = height;

else max = width;

//Se realiza la eliminación de jewels, registrando su posición (x, y)

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

//Valores inicalizads a -1 por la posibilidad de que no se pueda eliminar nada

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

int potentialJewelsLeft = 0;

int potentialJewelsRight = 0;

//Explorar por la izquierda

if ((x - 1 + y\*width >= 0) && table[x - 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x - i + y\*width >= 0) && (x - i >= 0) && table[x - i + y\*width] == table[x + y\*width]) {

potentialJewelsLeft++;

i++;

}

}

//Explorar por la derecha

if ((x + 1 + y\*width <= size) && table[x + 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x + i + y\*width <= size) && (x + i < width) && table[x + i + y\*width] == table[x + y\*width]) {

potentialJewelsRight++;

i++;

}

}

//Posibilidad de realizar eliminación horizontal

if (1 + potentialJewelsLeft + potentialJewelsRight >= 3) {

int stride = 0;

for (int j = potentialJewelsLeft; j >= (1); j--) {

erasedJewels[stride] = x - j;

erasedJewels[stride + 1] = y;

stride += 2;

}

erasedJewels[potentialJewelsLeft \* 2] = x;

erasedJewels[potentialJewelsLeft \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsRight; k++) {

erasedJewels[stride + potentialJewelsLeft \* 2] = x + k;

erasedJewels[stride + potentialJewelsLeft \* 2 + 1] = y;

stride += 2;

}

}

else { //Análisis de la vertical

int potentialJewelsAbove = 0;

int potentialJewelsBelow = 0;

//Explorar por debajo

if ((x + (y - 1)\*width >= 0) && table[x + (y - 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y - i)\*width >= 0) && table[x + (y - i)\*width] == table[x + y\*width]) {

potentialJewelsBelow++;

i++;

}

}

//Explorar por arriba

if ((x + 1 + y\*width <= size) && table[x + (y + 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y + i)\*width <= size) && table[x + (y + i)\*width] == table[x + y\*width]) {

potentialJewelsAbove++;

i++;

}

}

//Eliminación en la vertical

if (1 + potentialJewelsBelow + potentialJewelsAbove >= 3) {

int stride = 0;

for (int j = potentialJewelsBelow; j >= (1); j--) {

erasedJewels[stride] = x;

erasedJewels[stride + 1] = y - j;

stride += 2;

}

erasedJewels[potentialJewelsBelow \* 2] = x;

erasedJewels[potentialJewelsBelow \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsAbove; k++) {

erasedJewels[stride + potentialJewelsBelow \* 2] = x;

erasedJewels[stride + potentialJewelsBelow \* 2 + 1] = y + k;

stride += 2;

}

}

}

//Se ejecuta la eliminación

eraseJewels(table, erasedJewels, difficulty, width, height, TILE\_WIDTH, globalState);

free(erasedJewels);

}

//Desplaza la jewel seleccionada en la dirección indicada. Asume que el movimiento es posible

void swapSpots(float\* table, int jewel1X, int jewel1Y, int direction, int width, int height, int selection, int difficulty, int TILE\_WIDTH, curandState\* globalState) {

int jewel2X = jewel1X;

int jewel2Y = jewel1Y;

switch (direction)

{

case 1: //Arriba

{

jewel2Y += 1;

break;

}

case 2: //Abajo

{

jewel2Y -= 1;

break;

}

case 3: //Izquierda

{

jewel2X -= 1;

break;

}

case 4: //Derecha

{

jewel2X += 1;

break;

}

}

int aux1;

aux1 = table[jewel2X + jewel2Y\*width];

table[jewel2X + jewel2Y\*width] = table[jewel1X + jewel1Y\*width];

table[jewel1X + jewel1Y\*width] = aux1;

//Comprueba si hay eliminaciones posibles

manualTableAnalysis(difficulty, table, width, height, jewel2X, jewel2Y, TILE\_WIDTH, globalState);

}

//Determina y ejecuta el mejor movimiento posible

void automaticTableAnalysis(int difficulty, float\* table, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

float \*table\_d;

float \*aux\_d;

float \*aux;

//Tamaño del tablero en memoria

int size = width \* height \* sizeof(float);

int tam = width \* height;

int max = 0;

if (height >= width) max = height;

else max = width;

//Se evaluan las dos coordenadas de cada posición a eliminar

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

aux = (float\*)malloc(size);

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

for (int p = 0; p < tam; p++) {

aux[p] = 1;

}

//Envía el tablero a GPU

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

//Envía una variable auxiliar para conteo a GPU

cudaMalloc((void\*\*)&aux\_d, size);

cudaMemcpy(aux\_d, aux, size, cudaMemcpyHostToDevice);

//Cálculo de las dimensiones en base a TILE\_WIDTH

int wdth = ceil(((double)width) / TILE\_WIDTH);

int hght = ceil(((double)height) / TILE\_WIDTH);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

//Ejecución de kernel

automaticTableAnalysisKernel <<<dimGrid, dimBlock >>> (table\_d, aux\_d, difficulty, width, height, TILE\_WIDTH);

//Recupera el resultado de GPU a host

cudaMemcpy(aux, aux\_d, size, cudaMemcpyDeviceToHost);

int bestX = 0;

int bestY = 0;

int valor\_mejor = 0;

//Se exploran todos los movimientos en busca del mejor

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

if (aux[x + y\*width] > valor\_mejor) {

valor\_mejor = aux[x + y\*width];

bestX = x;

bestY = y;

}

}

}

//Se ejecuta el mejor movimiento

if (valor\_mejor >= 3) {

swapSpots(table, bestX, bestY, 4, width, height, 1, difficulty, TILE\_WIDTH, globalState);

}

free(aux);

free(erasedJewels);

cudaFree(table\_d);

cudaFree(aux\_d);

}

bool preload(int& width, int& height, int& difficulty, char\* file)

{

std::ifstream fwidth("width.txt");

std::ifstream fheight("height.txt");

std::ifstream fdifficulty("difficulty.txt");

std::ifstream loadF(file);

if (!fwidth.is\_open())

{

std::cout << "ERROR: no existe un archivo save." << std::endl;

return false;

}

if (!fheight.is\_open())

{

std::cout << "ERROR: no existe un archivo save." << std::endl;

return false;

}

if (!fdifficulty.is\_open())

{

std::cout << "ERROR: no existe un archivo save." << std::endl;

return false;

}

if (!loadF.is\_open())

{

std::cout << "ERROR: no existe un archivo save." << std::endl;

return false;

}

fwidth >> width;

fheight >> height;

fdifficulty >> difficulty;

fwidth.close();

fheight.close();

fdifficulty.close();

loadF.close();

return true;

}

void load(int width, int height, float\* table, char\* file)

{

int aux;

char\* array = (char\*)malloc(width\*height + 1);

std::ifstream loadF(file);

loadF.getline(array, width\*height + 1);

for (int i = 0; i < width\*height; i++)

{

aux = (array[i] - 48);

table[i] = (float)aux;

}

free(array);

loadF.close();

}

void save(float\* table, int width, int height, int difficulty, char\* file)

{

//Sistema de guardado

std::ofstream savedFile;

std::ofstream fileWidth;

std::ofstream fileHeight;

std::ofstream fileDifficulty;

savedFile.open(file);

fileWidth.open("width.txt");

fileHeight.open("height.txt");

fileDifficulty.open("difficulty.txt");

// Vacía los archivos de guardado

savedFile.clear();

fileWidth.clear();

fileHeight.clear();

fileDifficulty.clear();

// Escribe anchura, altura y dificultad

fileWidth << width;

fileHeight << height;

fileDifficulty << difficulty;

// Escribe el tablero al correspondiente archivo

for (int index = 0; index < width\*height; index++)

{

savedFile << table[index];

}

savedFile.close();

fileWidth.close();

fileHeight.close();

fileDifficulty.close();

}

// Función que elimina una fila completa

\_\_global\_\_ void rowBomb(float\* table, int width, int height, int difficulty, int row, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

float aux;

if ((trow + row) < height)

{

if (tcolumn < width)

{

if ((trow + row + 1) == height)

{

table[(trow + row)\*width + tcolumn] = createJewel(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

aux = table[(trow + row + 1)\*width + tcolumn];

table[(trow + row)\*width + tcolumn] = aux;

}

}

}

}

// Función que elimina una columna completa

\_\_global\_\_ void columnBomb(float\* table, int width, int height, int difficulty, int column, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH +threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height)

{

if ((column - tcolumn) >= 0)

{

if ((column - tcolumn - 1) < 0)

{

table[(trow\*width) + (column - tcolumn)] = createJewel(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

table[(trow\*width) + (column - tcolumn)] = table[(trow\*width) + (column - tcolumn - 1)];

}

}

}

}

\_\_global\_\_ void pivotBombGPU(float\* table, int width, int height, int row, int column)

{

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < 3)

{

if (tcolumn < 3)

{

table[(row + 1 - tcolumn)\*width + (column - 1 + trow)] = table[((row + 1) - trow)\*width + ((column + 1) - tcolumn)];

}

}

}

\_\_global\_\_ void pivotBomb(float\* table\_d, int width, int height, int TILE\_WIDTH)

{

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height && tcolumn < width) {

if (!((trow - 1) < 0 || (trow + 1) >= height || (tcolumn - 1) < 0 || (tcolumn + 1) >= width))

{

if (trow % 3 == 1 && tcolumn % 3 == 1)

{

dim3 dimBlock(3, 3);

dim3 dimGrid(1, 1);

pivotBombGPU <<<dimGrid, dimBlock >>> (table\_d, width, height, trow, tcolumn);

}

}

}

}

int main(int argc, char\*\* argv) {

int width;

int height;

int difficulty;

char mode;

int size;

char savedFile[9] = "save.txt";

int selection;

float\* table;

float\* table\_d;

curandState\* devStates;

bool playing = true;

//Configuración inicial del juego

if (argc == 1)

{

std::cout << "width del table: ";

std::cin >> width;

std::cout << "height del table: ";

std::cin >> height;

std::cout << "Elija difficulty: \n1.-Facil \n2.-Media \n3.-Dificil\n";

std::cin >> difficulty;

std::cout << "Automatico? 1.-SI 2.-NO\n";

std::cin >> selection;

}

else

{

mode = argv[1][1];

difficulty = atoi(argv[2]);

width = atoi(argv[3]);

height = atoi(argv[4]);

switch (mode) {

case 'a': {selection = 1; break; }

case 'm': {selection = 2; break; }

default: printf("Valor no valido.\n"); return -1;

}

}

size = width\*height;

//Tamaño de los bloques que se van a emplear.

int TILE\_WIDTH = getTileWidth(width, height);

if (TILE\_WIDTH == -1)

{

printf("ERROR: TILE\_WIDTH no valido");

return 0;

}

//Creación de las dimensiones de la tesela

int wdth = ceil(((float)width) / TILE\_WIDTH);

int hght = ceil(((float)height) / TILE\_WIDTH);

//Inicialización del random de CUDA

cudaMalloc(&devStates, size \* sizeof(curandState));

setup\_kernel <<< 1, size >>> (devStates, unsigned(time(NULL)));

//Creación de punteros al tablero y la versión en GPU del mismo

table = (float\*)malloc(size \* sizeof(float));

cudaMalloc((void\*\*)&table\_d, size \* sizeof(float));

//Inicialización del tablero

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

rndmTableInit <<<dimGrid, dimBlock >>>(table\_d, difficulty, width, height, TILE\_WIDTH, devStates);

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

//Bucle de juego: se muestran el estado actual y las opciones disponibles, y se procesa la selección del usuario.

while (playing) {

printTable(table, width, height);

int jewel1X = 0;

int jewel1Y = 0;

int command = 0;

std::cout << "Acción a realizar:"<<std::endl;

std::cout << "1.-\tIntercambiar Jewels"<<std::endl;

std::cout << "2.-\tGuardar partida"<<std::endl;

std::cout << "3.-\tCargar partida"<<std::endl;

std::cout << "9.-\tUsar una bomba"<<std::endl;

std::cout << "0.-\tSalir"<<std::endl;

std::cout << "Inserte seleccion: ";

std::cin >> command;

switch (command) {

/\* EXIT \*/

case 0: {

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}

//El jugador desea desplazar una jewel. El juego consulta la jewel a desplazar y la dirección, confirmando que el movimiento

//seleccionado es válido.

case 1: {

if (selection == 2)

{

std::cout << "Posicion de la primera jewel a intercambiar (empiezan en 0)\n";

std::cout << "column: ";

std::cin >> jewel1X;

std::cout << "row: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion erronea.\n");

continue;

}

int direction = 0;

std::cout << "Posicion de la jewel (el primer valor es 0):"<<std::endl;

std::cout << "\tColumna: ";

std::cin >> jewel1X;

std::cout << "\tFila: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion invalida.\n");

continue;

}

std::cout << "Direccion del movimiento:<<"<<std::endl<<"\t1.-\tArriba"<<std::endl<<"\t2.-\tAbajo"<<std::endl<<"\t3.-\tIzquierda"<<std::endl<<"\t4.-\tDerecha"<<std::endl;

std::cin >> direction;

if (direction > 4 && direction > 1) {

printf("Movimiento invalido.\n");

continue;

}

else {

switch (direction)

{

case 1: //Arriba

{

if (jewel1Y == height)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 2: //Abajo

{

if (jewel1Y == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 3: //Izquierda

{

if (jewel1X == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 4: //Derecha

{

if (jewel1X == width - 1)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

}

}

//Intercambio entre las jewels

swapSpots(table, jewel1X, jewel1Y, direction, width, height, selection, difficulty, TILE\_WIDTH, devStates);

}

else if (selection == 1)

{

//Modo automático activado, se procede a la selección y ejecución de movimiento

automaticTableAnalysis(difficulty, table, width, height, TILE\_WIDTH, devStates);

}

break;

}

//Opción guardar: el estado de la partida actual se escribe a un fichero externo.

case 2: {

save(table, width, height, difficulty, savedFile);

std::cout << "Guardado completado"<<std::endl;

break;

}

//Opción cargar: se lee el estado de la partida guardada y se recrea

case 3: {

//El juego prepara la carga del tablero

int found = preload(width, height, difficulty, savedFile);

size = width\*height;

if (found)

{

free(table);

table = (float\*)malloc(size \* sizeof(float));

// Carga efectiva del tablero

load(width, height, table, savedFile);

std::cout << "Juego automatico?"<<std::endl<<"\t1.-\tSI"<<std::endl<<"\t2.-\tNO"<<std::endl;

std::cin >> selection;

std::cout << "Se ha cargado el estado del tablero:"<<std::endl;

}

else {

std::cout << "No hay partidas guardadas."<<std::endl;

}

break;

}

case 9: {

int bomb = 0;

int row = 0; int column = 0;

std::cout << "Elija una bomba:";

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

/\*

Selección del tipo de bomba y ejecución del mismo. Los tipos disponibles depende del nivel de dificultad seleccionado,

de acuerdo a las condiciones establecidas en el documento de la práctica.

\*/

switch (difficulty) {

case 1: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb != 1)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 2)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2:

{

std::cout << "Columna: ";

std::cin >> column;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

columnBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, TILE\_WIDTH, devStates);

break;

}

}

break;

}

case 3: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "3.-\tBomba de rotacion 3x3"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 3)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

rowBomb <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2:

{

std::cout << "Columna: ";

std::cin >> column;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

columnBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, TILE\_WIDTH, devStates);

break;

}

case 3:

{

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

pivotBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, TILE\_WIDTH);

break;

}

}

break;

}

}

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

break;

}

}

}

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}

# Versión en GPU usando memoria compartida

#include <stdio.h>

#include <cuda\_runtime.h>

#include <iostream>

#include <cstdlib>

#include <curand.h>

#include <curand\_kernel.h>

#include <ctime>

#include <fstream>

//Compruebas las propiedades de la tarjeta gráfica para generar una tesela adecuada, al tiempo que evalua el tamaño del tablero.

int getTileWidth(int width, int height) {

float minSize = 0;

if (width > height) minSize = width;

else minSize = height;

cudaDeviceProp properties;

cudaGetDeviceProperties(&properties, 0);

int maxThreads = properties.maxThreadsPerBlock;

if (width == height) { //En caso de una matriz cuadrada, queremos tener más de un bloque

if (minSize / 32 > 1 && maxThreads == 1024) { //El bloque solo podrá ser de 32x32 si tiene 1024 hilos o más

return 32;

}

else if (minSize / 16 > 1) {

return 16;

}

else if (minSize / 8 > 1) {

return 8;

}

else if (minSize / 4 > 1) {

return 4;

}

else if (minSize / 2 > 1) {

return 2;

}

}

else {

if (minSize / 32 >= 1 && maxThreads == 1024) {

return 32;

}

else if (minSize / 16 >= 1) {

return 16;

}

else if (minSize / 8 >= 1) {

return 8;

}

else if (minSize / 4 >= 1) {

return 4;

}

else if (minSize / 2 >= 1) {

return 2;

}

}

return -1;

}

// Función que inicializa la semilla del generador de números aleatorios de cuda

\_\_global\_\_ void setup\_kernel(curandState \* state, unsigned long seed)

{

int id = threadIdx.x;

curand\_init(seed, id, 0, &state[id]);

}

//Función que crea una jewel utilizando globalState

\_\_device\_\_ float generate(curandState\* globalState, int ind)

{

curandState localState = globalState[ind];

float rndm = curand\_uniform(&localState);

globalState[ind] = localState;

return rndm;

}

// Función que crea una jewel aleatoria en CUDA

\_\_device\_\_ int createJewel(curandState\* globalState, int ind, int difficulty)

{

switch (difficulty) {

case 1:

{

return (int)1 + generate(globalState, ind) \* 4;

}

case 2: {

return (int)1 + generate(globalState, ind) \* 6;

}

case 3: {

return (int)1 + generate(globalState, ind) \* 8;

}

}

return -1;

}

// Función que rellena inicialmente el tablero

\_\_global\_\_ void randomTableInit(float \*table, int difficulty, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height)

{

if (tcolumn < width)

{

table[trow\*width + tcolumn] = createJewel(globalState, trow \* width + tcolumn, difficulty);

}

}

}

void printTable(float\* table, int width, int height) {

for (int i = height - 1; i >= 0; i--) {

printf("\n");

for (int j = 0; j < width; j++) {

printf("%d ", (int)table[j + i\*width]);

}

}

printf("\n");

}

//Función que, en base a las coordenadas de su hilo de ejecución, determina si la jewel correspondiente ha de ser eliminada y hace descender las filas

//correspondientes en base a ello. Utiliza una copia del tablero para evitar condiciones de carrera.

\_\_global\_\_ void eraseJewelsKernel(float\* table\_d, float\* auxTable\_d, float\* erasedJewels\_d, int difficulty, int width, int height, int end, int TILE\_WIDTH, curandState\* globalState) {

int tx = threadIdx.x;

int ty = threadIdx.y;

int block\_x = blockIdx.x;

int block\_y = blockIdx.y;

//Calcula la posición real dentro del tablero

tx += block\_x \* TILE\_WIDTH;

ty += block\_y \* TILE\_WIDTH;

if (erasedJewels\_d[0] != erasedJewels\_d[2] && tx >= erasedJewels\_d[0] && tx <= erasedJewels\_d[end - 2] && ty >= erasedJewels\_d[1]) {

if (ty + 1 < height) {

float value = auxTable\_d[tx + (ty + 1)\*width];

table\_d[tx + (ty)\*(width)] = value;

}

else {

table\_d[tx + ty\*width] = createJewel(globalState, tx + ty\*width, difficulty);

}

}

else {

if (ty < height && tx == erasedJewels\_d[0] && ty > erasedJewels\_d[1]) {

float value = auxTable\_d[tx + (ty)\*width];

table\_d[tx + (ty - end / 2)\*(width)] = value;

}

if (ty >= height - end / 2 && ty < height && tx == erasedJewels\_d[0]) {

table\_d[tx + (ty)\*width] = createJewel(globalState, tx + ty\*width, difficulty);

}

}

}

//Función que prepara y ejecuta la eliminación de jewels en GPU.

void eraseJewels(float\* table, float\* erasedJewels, int difficulty, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

float \*table\_d;

float \*erasedJewels\_d;

float \*auxTable\_d;

int size = width \* height \* sizeof(float);

int max = 0;

//Determina cuántas jewels se pueden, como mucho, eliminar en una dirección.

if (height >= width) max = height;

else max = width;

//Envía a GPU el tablero y una copia

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

cudaMalloc((void\*\*)&auxTable\_d, size);

cudaMemcpy(auxTable\_d, table, size, cudaMemcpyHostToDevice);

//Envía las jewels a eliminar a GPU. Se necesita reservar dos veces la cantidad de memoria ocupada

//por max\*sizeof(float) porque cada jewel se identifica mediante dos cifras.

cudaMalloc((void\*\*)&erasedJewels\_d, 2 \* max \* sizeof(float));

cudaMemcpy(erasedJewels\_d, erasedJewels, 2 \* max \* sizeof(float), cudaMemcpyHostToDevice);

int end = 0;

bool altered = false;

//Calcula cuál es el último valor escrito de entre todas las jewels a eliminar

for (int i = 0; i < max \* 2; i++) {

if (erasedJewels[i] < 0) {

end = i;

altered = true;

break;

}

}

if (!altered) end = max \* 2;

//Calcula las dimensiones en función de TILE\_WIDTH

int wdth = ceil(((double)width) / TILE\_WIDTH);

int hght = ceil(((double)height) / TILE\_WIDTH);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

eraseJewelsKernel <<<dimGrid, dimBlock>>> (table\_d, auxTable\_d, erasedJewels\_d, difficulty, width, height, end, TILE\_WIDTH, globalState);

//Recuperación del nuevo tablero

cudaMemcpy(table, table\_d, size, cudaMemcpyDeviceToHost);

cudaFree(table\_d);

cudaFree(erasedJewels\_d);

cudaFree(auxTable\_d);

}

//Análisis del tablero tras un movimiento manual, en base a las nuevas coordenadas de la jewel desplazada

void manualTableAnalysis(int difficulty, float\* table, int width, int height, int x, int y, int TILE\_WIDTH, curandState\* globalState) {

int max = 0;

int size = width\*height;

if (height >= width) max = height;

else max = width;

//Se eliminan todas las jewels posibles, almacenando su posición (x,y)

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

//Iniciamos todos los valores a -1 para identificar las posiciones modificadas

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

int potentialJewelsLeft = 0;

int potentialJewelsRight = 0;

//Explorar por la izquierda

if ((x - 1 + y\*width >= 0) && table[x - 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x - i + y\*width >= 0) && (x - i >= 0) && table[x - i + y\*width] == table[x + y\*width]) {

potentialJewelsLeft++;

i++;

}

}

//Explorar por la derecha

if ((x + 1 + y\*width <= size) && table[x + 1 + y\*width] == table[x + y\*width]) {

int i = 1;

while ((x + i + y\*width <= size) && (x + i < width) && table[x + i + y\*width] == table[x + y\*width]) {

potentialJewelsRight++;

i++;

}

}

//Eliminación horizontal

if (1 + potentialJewelsLeft + potentialJewelsRight >= 3) {

int stride = 0;

for (int j = potentialJewelsLeft; j >= (1); j--) {

erasedJewels[stride] = x - j;

erasedJewels[stride + 1] = y;

stride += 2;

}

erasedJewels[potentialJewelsLeft \* 2] = x;

erasedJewels[potentialJewelsLeft \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsRight; k++) {

erasedJewels[stride + potentialJewelsLeft \* 2] = x + k;

erasedJewels[stride + potentialJewelsLeft \* 2 + 1] = y;

stride += 2;

}

}

else { //Análisis vertical

int potentialJewelsAbove = 0;

int potentialJewelsBelow = 0;

//Explorar abajo

if ((x + (y - 1)\*width >= 0) && table[x + (y - 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y - i)\*width >= 0) && table[x + (y - i)\*width] == table[x + y\*width]) {

potentialJewelsBelow++;

i++;

}

}

//Explorar arriba

if ((x + 1 + y\*width <= size) && table[x + (y + 1)\*width] == table[x + y\*width]) {

int i = 1;

while ((x + (y + i)\*width <= size) && table[x + (y + i)\*width] == table[x + y\*width]) {

potentialJewelsAbove++;

i++;

}

}

//Eliminación vertical

if (1 + potentialJewelsBelow + potentialJewelsAbove >= 3) {

int stride = 0;

for (int j = potentialJewelsBelow; j >= (1); j--) {

erasedJewels[stride] = x;

erasedJewels[stride + 1] = y - j;

stride += 2;

}

erasedJewels[potentialJewelsBelow \* 2] = x;

erasedJewels[potentialJewelsBelow \* 2 + 1] = y;

stride = 2;

for (int k = 1; k <= potentialJewelsAbove; k++) {

erasedJewels[stride + potentialJewelsBelow \* 2] = x;

erasedJewels[stride + potentialJewelsBelow \* 2 + 1] = y + k;

stride += 2;

}

}

}

//Se ejecuta la eliminación propiamente dicha

eraseJewels(table, erasedJewels, difficulty, width, height, TILE\_WIDTH, globalState);

free(erasedJewels);

}

//Desplaza la jewel seleccionada en la dirección indicada

void swapSpots(float\* table, int jewel1X, int jewel1Y, int direction, int width, int height, int selection, int difficulty, int TILE\_WIDTH, curandState\* globalState) {

int jewel2X = jewel1X;

int jewel2Y = jewel1Y;

switch (direction)

{

case 1: //Arriba

{

jewel2Y += 1;

break;

}

case 2: //Abajo

{

jewel2Y -= 1;

break;

}

case 3: //Izquierda

{

jewel2X -= 1;

break;

}

case 4: //Derecha

{

jewel2X += 1;

break;

}

}

int aux1;

aux1 = table[jewel2X + jewel2Y\*width];

table[jewel2X + jewel2Y\*width] = table[jewel1X + jewel1Y\*width];

table[jewel1X + jewel1Y\*width] = aux1;

//Determina si se pueden realizar eliminaciones

manualTableAnalysis(difficulty, table, width, height, jewel2X, jewel2Y,TILE\_WIDTH, globalState);

}

//Función que, en base a las coordenadas de su hilo de ejecución, determina si la jewel correspondiente ha de ser eliminada y hace descender las filas

//correspondientes en base a ello. Utiliza una copia del tablero para evitar condiciones de carrera.

\_\_global\_\_ void automaticTableAnalysisKernel(float \*table\_d, float \*aux\_d, int difficulty, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

int tx = threadIdx.x;

int ty = threadIdx.y;

int block\_x = blockIdx.x;

int block\_y = blockIdx.y;

//Determina la posición real dentro del tablero

tx += block\_x \* TILE\_WIDTH;

ty += block\_y \* TILE\_WIDTH;

//Manejo del array dinámico en memoria compartida mediante hilos

extern \_\_shared\_\_ float sharedTable[];

sharedTable[tx + ty\*width] = aux\_d[tx + ty\*width];

\_\_syncthreads();

int potentialJewelsRight = 0;

//Explora la derecha

if ((tx + 2) < width) {

if (((tx + 2) + ty\*width <= height\*width) && table\_d[tx + 2 + ty\*width] == table\_d[tx + ty\*width]) {

int i = 2;

while ((tx + i + ty\*width <= height\*width) && table\_d[tx + i + ty\*width] == table\_d[tx + ty\*width]) {

potentialJewelsRight++;

i++;

}

sharedTable[tx + ty\*width] = potentialJewelsRight + 1;

}

else {

sharedTable[tx + ty\*width] = 1;

}

}

else {

sharedTable[tx + ty\*width] = 1;

}

//Todos los hilos esperan hasta que elo cálculo está completo

\_\_syncthreads();

aux\_d[tx + ty\*width] = sharedTable[tx + ty\*width];

}

//Determina el mejor movimiento posible y lo ejecuta

void automaticTableAnalysis(int difficulty, float\* table, int width, int height, int TILE\_WIDTH, curandState\* globalState) {

float \*table\_d;

float \*aux\_d;

float \*aux;

//Tamaño del tablero en memoria

int size = width \* height \* sizeof(float);

int size1 = width \* height;

int max = 0;

if (height >= width) max = height;

else max = width;

float\* erasedJewels = (float\*)malloc(2 \* max \* sizeof(float));

aux = (float\*)malloc(size);

for (int i = 0; i < max; i++) {

erasedJewels[i] = -1;

}

for (int p = 0; p < size1; p++) {

aux[p] = 1;

}

//Envía el tablero a GPU

cudaMalloc((void\*\*)&table\_d, size);

cudaMemcpy(table\_d, table, size, cudaMemcpyHostToDevice);

//Envía una variable auxiliar para conteo a GPU

cudaMalloc((void\*\*)&aux\_d, size);

cudaMemcpy(aux\_d, aux, size, cudaMemcpyHostToDevice);

//Calcula las medidas en función de TILE\_WIDTH

int wdth = ceil(((double)width) / TILE\_WIDTH);

int hght = ceil(((double)height) / TILE\_WIDTH);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

//Ejecuta el kernel

automaticTableAnalysisKernel <<<dimGrid, dimBlock, 2\*width \* height \* sizeof(float) >>> (table\_d, aux\_d, difficulty, width, height, TILE\_WIDTH, globalState);

//Recupera el resultado

cudaMemcpy(aux, aux\_d, size, cudaMemcpyDeviceToHost);

int bestX = 0;

int bestY = 0;

int bestValue = 0;

//Se determina el mejor movimiento

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

if (aux[x + y\*width] > bestValue) {

bestValue = aux[x + y\*width];

bestX = x;

bestY = y;

}

}

}

//Se ejecuta el mejor movimiento

if (bestValue >= 3) {

swapSpots(table, bestX, bestY, 4, width, height, 1, difficulty, TILE\_WIDTH, globalState);

}

free(aux);

free(erasedJewels);

cudaFree(table\_d);

cudaFree(aux\_d);

}

bool preload(int& width, int& height, int& difficulty, char\* file)

{

std::ifstream loadF(file);

char siz[4];

if (!loadF.is\_open())

{

std::cout << "ERROR: no existe un archivo save." << std::endl;

return false;

}

loadF.getline(siz, 4);

width = (int)siz[0] - 48;

height = (int)siz[1] - 48;

difficulty = (int)siz[2] - 48;

loadF.close();

return true;

}

void load(int width, int height, float\* table, char\* file)

{

char\* array = (char\*)malloc(width\*height + 1 + 3);

std::ifstream loadF(file);

loadF.getline(array, (width\*height + 1 + 3));

for (int i = 0; i < width\*height; i++)

{

table[i] = array[i + 3] - 48;

}

free(array);

loadF.close();

}

void save(float\* table, int width, int height, int difficulty, char\* file)

{

//Sistema de guardado

std::ofstream savedFile;

savedFile.open(file);

savedFile.clear();

//Almacenar anchura, altura y dificultad

savedFile << width;

savedFile << height;

savedFile << difficulty;

//Almacenar tablero

for (int index = 0; index < width\*height; index++)

{

savedFile << table[index];

}

savedFile.close();

}

//Función que elimina una fila completa

\_\_global\_\_ void bombRow(float\* table, int width, int height, int difficulty, int row, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

float aux;

if ((trow + row) < height)

{

if (tcolumn < width)

{

if ((trow + row + 1) == height)

{

table[(trow + row)\*width + tcolumn] = createJewel(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

aux = table[(trow + row + 1)\*width + tcolumn];

table[(trow + row)\*width + tcolumn] = aux;

}

}

}

}

// Función que elimina una columna completa

\_\_global\_\_ void bombColumn(float\* table, int width, int height, int difficulty, int column, int TILE\_WIDTH, curandState\* globalState) {

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height)

{

if ((column - tcolumn) >= 0)

{

if ((column - tcolumn - 1) < 0)

{

table[(trow\*width) + (column - tcolumn)] = createJewel(globalState, (trow \* 3 + tcolumn), difficulty);

}

else {

table[(trow\*width) + (column - tcolumn)] = table[(trow\*width) + (column - tcolumn - 1)];

}

}

}

}

\_\_global\_\_ void pivotBombGPU(float\* table, int width, int height, int row, int column)

{

\_\_shared\_\_ int aux[9];

int trow = threadIdx.y;

int tcolumn = threadIdx.x;

if (trow < 3)

{

if (tcolumn < 3)

{

//Uso de la memoria compartida

aux[trow + tcolumn \* 3] = table[((row + 1) - trow)\*width + ((column + 1) - tcolumn)];

\_\_syncthreads();

table[((row + 1) - trow)\*width + ((column - 1) + tcolumn)] = aux[trow \* 3 + tcolumn];

}

}

}

\_\_global\_\_ void pivotBomb(float\* table\_d, int width, int height, int TILE\_WIDTH)

{

int trow = blockIdx.y\*TILE\_WIDTH + threadIdx.y;

int tcolumn = blockIdx.x\*TILE\_WIDTH + threadIdx.x;

if (trow < height && tcolumn < width) {

if (!((trow - 1) < 0 || (trow + 1) >= height || (tcolumn - 1) < 0 || (tcolumn + 1) >= width))

{

if (trow % 3 == 1 && tcolumn % 3 == 1)

{

dim3 dimBlock(3, 3);

dim3 dimGrid(1, 1);

pivotBombGPU <<<dimGrid, dimBlock >>> (table\_d, width, height, trow, tcolumn);

}

}

}

}

int main(int argc, char\*\* argv) {

int width;

int height;

int difficulty;

char mode;

int size;

char savedFile[9] = "save.txt";

int selection;

float\* table;

float\* table\_d;

curandState\* devStates;

bool playing = true;

//Configuración inicial del juego

if (argc == 1)

{

std::cout << "width del table: ";

std::cin >> width;

std::cout << "height del table: ";

std::cin >> height;

std::cout << "Elija difficulty: \n1.-Facil \n2.-Media \n3.-Dificil\n";

std::cin >> difficulty;

std::cout << "Automatico? 1.-SI 2.-NO\n";

std::cin >> selection;

}

else

{

mode = argv[1][1];

difficulty = atoi(argv[2]);

width = atoi(argv[3]);

height = atoi(argv[4]);

switch (mode) {

case 'a': {selection = 1; break; }

case 'm': {selection = 2; break; }

default: printf("Valor no valido.\n"); return -1;

}

}

size = width\*height;

//Tamaño de los bloques que se van a emplear.

int TILE\_WIDTH = obtenerTileWidth(width, height);

if (TILE\_WIDTH == -1)

{

printf("ERROR: TILE\_WIDTH no valido");

return 0;

}

//Creación de las dimensiones de la tesela

int wdth = ceil(((float)width) / TILE\_WIDTH);

int hght = ceil(((float)height) / TILE\_WIDTH);

//Inicialización del random de CUDA

cudaMalloc(&devStates, size \* sizeof(curandState));

setup\_kernel <<< 1, size >>> (devStates, unsigned(time(NULL)));

//Creación de punteros al tablero y la versión en GPU del mismo

table = (float\*)malloc(size \* sizeof(float));

cudaMalloc((void\*\*)&table\_d, size \* sizeof(float));

//Inicialización del tablero

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

randomTableInit <<<dimGrid, dimBlock >>>(table\_d, difficulty, width, height, TILE\_WIDTH, devStates);

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

//Bucle de juego: se muestran el estado actual y las opciones disponibles, y se procesa la selección del usuario.

while (playing) {

printTable(table, width, height);

int jewel1X = 0;

int jewel1Y = 0;

int command = 0;

std::cout << "Acción a realizar:"<<std::endl;

std::cout << "1.-\tIntercambiar Jewels"<<std::endl;

std::cout << "2.-\tGuardar partida"<<std::endl;

std::cout << "3.-\tCargar partida"<<std::endl;

std::cout << "9.-\tUsar una bomba"<<std::endl;

std::cout << "0.-\tSalir"<<std::endl;

std::cout << "Inserte seleccion: ";

std::cin >> command;

switch (command) {

/\* EXIT \*/

case 0: {

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}

//El jugador desea desplazar una jewel. El juego consulta la jewel a desplazar y la dirección, confirmando que el movimiento

//seleccionado es válido.

case 1: {

if (selection == 2)

{

std::cout << "Posicion de la primera jewel a intercambiar (empiezan en 0)\n";

std::cout << "column: ";

std::cin >> jewel1X;

std::cout << "row: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion erronea.\n");

continue;

}

int direction = 0;

std::cout << "Posicion de la jewel (el primer valor es 0):"<<std::endl;

std::cout << "\tColumna: ";

std::cin >> jewel1X;

std::cout << "\tFila: ";

std::cin >> jewel1Y;

if (!((jewel1X < width) && (jewel1X >= 0) && (jewel1Y < height) && (jewel1Y >= 0))) {

printf("Posicion invalida.\n");

continue;

}

std::cout << "Direccion del movimiento:<<"<<std::endl<<"\t1.-\tArriba"<<std::endl<<"\t2.-\tAbajo"<<std::endl<<"\t3.-\tIzquierda"<<std::endl<<"\t4.-\tDerecha"<<std::endl;

std::cin >> direction;

if (direction > 4 && direction > 1) {

printf("Movimiento invalido.\n");

continue;

}

else {

switch (direction)

{

case 1: //Arriba

{

if (jewel1Y == height)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 2: //Abajo

{

if (jewel1Y == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 3: //Izquierda

{

if (jewel1X == 0)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

case 4: //Derecha

{

if (jewel1X == width - 1)

{

printf("No se puede realizar el intercambio especificado.\n");

continue;

}

break;

}

}

}

//Intercambio entre las jewels

swapSpots(table, jewel1X, jewel1Y, direction, width, height, selection, difficulty, TILE\_WIDTH, devStates);

}

else if (selection == 1)

{

//Modo automático activado, se procede a la selección y ejecución de movimiento

automaticTableAnalysis(difficulty, table, width, height, TILE\_WIDTH, devStates);

}

break;

}

//Opción guardar: el estado de la partida actual se escribe a un fichero externo.

case 2: {

save(table, width, height, difficulty, savedFile);

std::cout << "Guardado completado"<<std::endl;

break;

}

//Opción cargar: se lee el estado de la partida guardada y se recrea

case 3: {

//El juego prepara la carga del tablero

int found = preload(width, height, difficulty, savedFile);

size = width\*height;

if (found)

{

free(table);

table = (float\*)malloc(size \* sizeof(float));

// Carga efectiva del tablero

load(width, height, table, savedFile);

std::cout << "Juego automatico?"<<std::endl<<"\t1.-\tSI"<<std::endl<<"\t2.-\tNO"<<std::endl;

std::cin >> selection;

std::cout << "Se ha cargado el estado del tablero:"<<std::endl;

}

else {

std::cout << "No hay partidas guardadas."<<std::endl;

}

break;

}

case 9: {

int bomb = 0;

int row = 0; int column = 0;

std::cout << "Elija una bomba:";

cudaMemcpy(table\_d, table, size \* sizeof(float), cudaMemcpyHostToDevice);

/\*

Selección del tipo de bomba y ejecución del mismo. Los tipos disponibles depende del nivel de dificultad seleccionado,

de acuerdo a las condiciones establecidas en el documento de la práctica.

\*/

switch (difficulty) {

case 1: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb != 1)

{

printf("bomb erronea.\n");

continue;

}

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

bombRow <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 2)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

bombRow <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2:

{

std::cout << "Columna: ";

std::cin >> column;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

bombColumn <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, TILE\_WIDTH, devStates);

break;

}

}

break;

}

case 3: {

std::cout << "1.-\tBomba sobre fila"<<std::endl;

std::cout << "2.-\tBomba sobre columna"<<std::endl;

std::cout << "3.-\tBomba de rotacion 3x3"<<std::endl;

std::cout << "\tSeleccion: "<<std::endl;

std::cin >> bomb;

if (bomb < 1 && bomb > 3)

{

printf("Tipo de bomba inexistente.\n");

continue;

}

switch (bomb) {

case 1:

{

std::cout << "Fila: ";

std::cin >> row;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

bombRow <<<dimGrid, dimBlock >>> (table\_d, width, height, difficulty, row, TILE\_WIDTH, devStates);

break;

}

case 2:

{

std::cout << "Columna: ";

std::cin >> column;

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

bombColumn <<<dimGrid, dimBlock >>>(table\_d, width, height, difficulty, column, TILE\_WIDTH, devStates);

break;

}

case 3:

{

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH);

dim3 dimGrid(wdth, hght);

pivotBomb <<<dimGrid, dimBlock >>>(table\_d, width, height, TILE\_WIDTH);

break;

}

}

break;

}

}

cudaMemcpy(table, table\_d, size \* sizeof(float), cudaMemcpyDeviceToHost);

break;

}

}

}

free(table);

cudaFree(table\_d);

cudaFree(devStates);

return 0;

}