#include "include/kernel.cuh"

/\* ------------------------ \*/

/\* FUNCIONES DE DISPOSITIVO \*/

/\* ------------------------ \*/

/\*\*

\* Comprueba si es posible realizar un giro de 3x3 en la posición dada.

\*

\* @param posY

\* Coordenada Y del eje a comprobar.

\*

\* @param posX

\* Coordenada X del eje a comprobar.

\*

\* @param dimens

\* Dimensiones de la matriz a comprobar.

\*

\*

\* @return

\* true si es posible.

\* false si no lo es.

\*/

\_\_device\_\_ bool comprobar\_giro (int posY, int posX, Dim dimens)

{

int filas = dimens.filas,

cols = dimens.columnas;

/\* Comprueba los límites del eje de giro \*/

if ( ((posY + 1) >= filas)

|| ((posX + 1) >= cols) )

{

return false;

}

if(((posY - 1) == 0)

|| ((posY - 1) % 3) == 0)

{

/\* Posición correcta para el eje Y \*/

if(((posX - 1) == 0)

|| ((posX - 1) % 3) == 0)

{

/\* Posición correcta para el eje X \*/

return true;

}

}

return false;

}

/\*\*

\* Busca el primer elemento no vacío por encima de la posición especificada.

\* Además, este elemento se convierte a DIAMANTE\_VACIO.

\*

\* @param matriz

\* Matriz en la que se ha de buscar el elemento.

\*

\* @param fila\_ini

\* Fila del primer elemento a comprobar.

\*

\* @param columna

\* Columna a comprobar.

\*

\* @param dimens

\* Dimensiones de la matriz.

\*

\*

\* @return

\* El primer elemento encontrado, si había alguno.

\* -1 si no se encontró ningún elemento no vacío.

\*/

\_\_device\_\_ int buscar\_lleno (int \*matriz, int fila\_ini, int columna, Dim dimens)

{

int elem = -1,

fila = fila\_ini,

aux;

while ( (elem == -1)

&& (fila >= 0))

{

aux = (fila \* dimens.columnas) + columna;

if (matriz [aux] != DIAMANTE\_VACIO)

{

elem = matriz [aux];

matriz [aux] = DIAMANTE\_VACIO;

}

fila--;

}

return elem;

}

/\* ------- \*/

/\* NÚCLEOS \*/

/\* ------- \*/

/\*\*

\* Genera un número aleatorio en base a la secuencia especificada y al índice del hilo.

\*

\* @param semilla

\* Elemento inicial para generar la secuencia.

\*

\* @param resultado

\* Vector en el que se almacenarán los números generados.

\*

\* @param min

\* Límite inferior para generar un número (inclusivo).

\*

\* @param max

\* Límite superior para generar un número (inclusivo).

\*

\* @param dimens

\* Dimensiones de la matriz resultado.

\*/

\_\_global\_\_ void gen\_aleat\_cuda (unsigned long semilla,

int \*resultado,

const int min,

const int max,

const Dim dimens)

{

int rand\_int,

fila = blockIdx.y \* blockDim.y + threadIdx.y,

columna = blockIdx.x \* blockDim.x + threadIdx.x,

aux = (fila \* dimens.columnas) + columna;

curandState estado;

/\* Comprueba los límites de la matriz \*/

if ((fila <= dimens.filas)

&& (columna <= dimens.columnas))

{

curand\_init (semilla, aux, 0, &estado);

/\* El número se genera primero con coma flotante (ajustando los

límites como se haya especificado) y luego se convierte a entero.

Esto es más rápido que realizar la operación de módulo \*/

float rand\_f = curand\_uniform (&estado);

rand\_f \*= (max - min + 0.999999);

rand\_f += min;

/\* Convierte el float a entero \*/

rand\_int = \_\_float2int\_rz (rand\_f);

/\* Guarda el resultado \*/

resultado [aux] = rand\_int;

}

}

/\*\*

\* Mueve todos los elementos a la izquierda de fila\_bomba hacia su derecha. Cuando llega

\* al primer elemento, genera un nuevo elemento.

\*

\* @param semilla

\* Elemento inicial para generar la secuencia (para crear los

\* nuevos elementos).

\*

\* @param resultado

\* Vector que almacena la matriz que va a ser cambiada.

\*

\* @param min

\* Límite inferior para generar un número (inclusivo).

\*

\* @param max

\* Límite superior para generar un número (inclusivo).

\*

\* @param dimens

\* Dimensiones de la matriz resultado.

\*

\*

\* @param fila\_bomb

\* Fila a eliminar.

\*/

\_\_global\_\_ void eliminar\_fila\_cuda (unsigned long semilla,

int \*resultado,

const int min,

const int max,

const Dim dimens,

int fila\_bomba)

{

int columna = blockIdx.x \* blockDim.x + threadIdx.x,

i,

rand\_int;

curandState estado;

float rand\_f;

if ((columna >= dimens.columnas)

|| ( (blockIdx.y \* blockDim.y + threadIdx.y) != 0) )

{

return;

}

/\* Intercambia los elementos desde la fila actual hasta el principio \*/

for (i = fila\_bomba; i > 0; i--)

{

resultado [(i \* dimens.columnas) + columna]

= resultado [( (i - 1) \* dimens.columnas ) + columna];

}

/\* Genera el último elemento \*/

curand\_init (semilla, columna, 0, &estado);

/\* El número se genera primero con coma flotante (ajustando los

límites como se haya especificado) y luego se convierte a entero.

Esto es más rápido que realizar la operación de módulo \*/

rand\_f = curand\_uniform (&estado);

rand\_f \*= (max - min + 0.999999);

rand\_f += min;

/\* Convierte el float a entero \*/

rand\_int = \_\_float2int\_rz (rand\_f);

/\* Guarda el resultado \*/

resultado [columna] = rand\_int;

}

/\*\*

\* Mueve todos los elementos a la izquierda de fila\_bomba hacia su derecha. Cuando llega

\* al primer elemento, genera un nuevo elemento.

\*

\* @param semilla

\* Elemento inicial para generar la secuencia (para crear los

\* nuevos elementos).

\*

\* @param resultado

\* Vector que almacena la matriz que va a ser cambiada.

\*

\* @param min

\* Límite inferior para generar un número (inclusivo).

\*

\* @param max

\* Límite superior para generar un número (inclusivo).

\*

\* @param dimens

\* Dimensiones de la matriz resultado.

\*

\*

\* @param fila\_bomb

\* Fila a eliminar.

\*/

\_\_global\_\_ void eliminar\_columna\_cuda (unsigned long semilla,

int \*resultado,

const int min,

const int max,

const Dim dimens,

int col\_bomba)

{

int fila = blockIdx.y \* blockDim.y + threadIdx.y,

i,

rand\_int;

curandState estado;

float rand\_f;

if ((fila >= dimens.filas)

|| ( (blockIdx.x \* blockDim.x + threadIdx.x) != 0) )

{

return;

}

/\* Intercambia los elementos desde la fila actual hasta el principio \*/

for (i = col\_bomba; i > 0; i--)

{

resultado [(fila \* dimens.columnas) + i]

= resultado [(fila \* dimens.columnas ) + i - 1];

}

/\* Genera el último elemento \*/

curand\_init (semilla, fila, 0, &estado);

/\* El número se genera primero con coma flotante (ajustando los

límites como se haya especificado) y luego se convierte a entero.

Esto es más rápido que realizar la operación de módulo \*/

rand\_f = curand\_uniform (&estado);

rand\_f \*= (max - min + 0.999999);

rand\_f += min;

/\* Convierte el float a entero \*/

rand\_int = \_\_float2int\_rz (rand\_f);

/\* Guarda el resultado \*/

resultado [fila \* dimens.columnas] = rand\_int;

}

/\*\*

\* Gira todos los elementos posibles en grupos de 3x3 (bomba III).

\*

\* @param resultado

\* Vector que almacena la matriz que va a ser cambiada.

\*

\* @param dimens

\* Dimensiones de la matriz.

\*/

\_\_global\_\_ void girar\_matriz\_cuda (int \*resultado,

Dim dimens)

{

int fila = blockIdx.y \* blockDim.y + threadIdx.y,

columna = blockIdx.x \* blockDim.x + threadIdx.x,

posY = fila - 1,

posX = columna - 1,

aux;

if ((fila >= dimens.filas)

|| (columna >= dimens.columnas))

{

return;

}

if (comprobar\_giro (fila, columna, dimens))

{

/\* Se realizan los intercambios de manera manual \*/

aux = resultado [(posY \* dimens.columnas) + posX];

/\* ---- \*/

resultado [(posY \* dimens.columnas) + posX]

= resultado [( (posY + 2) \* dimens.columnas) + posX];

resultado [( (posY + 2) \* dimens.columnas) + posX]

= resultado [( (posY + 2) \* dimens.columnas) + posX + 2];

resultado [( (posY + 2) \* dimens.columnas) + posX + 2]

= resultado [(posY \* dimens.columnas) + posX + 2];

resultado [(posY \* dimens.columnas) + posX + 2] = aux;

/\* ---- \*/

aux = resultado [(posY \* dimens.columnas) + posX + 1];

resultado [(posY \* dimens.columnas) + posX + 1]

= resultado [( (posY + 1) \* dimens.columnas) + posX];

resultado [( (posY + 1) \* dimens.columnas) + posX]

= resultado [( (posY + 2) \* dimens.columnas) + posX + 1];

resultado [( (posY + 2) \* dimens.columnas) + posX + 1]

= resultado [( (posY + 1) \* dimens.columnas) + posX + 2];

resultado [( (posY + 1) \* dimens.columnas) + posX + 2] = aux;

}

}

/\*\*

\* Comprueba si la fila contiene elementos repetidos.

\*

\* @param matriz

\* Matriz con los valores actuales de los diamantes.

\*

\* @param dimens

\* Estructura con las dimensiones de la matriz.

\*

\* @param coincidencias

\* Matriz en la que se va a indicar si había alguna coincidencia.

\*/

\_\_global\_\_ void busar\_coinc\_cuda\_fila (int \*matriz,

Dim dimens,

int \*coincidencias)

{

int fila = blockIdx.y \* blockDim.y + threadIdx.y,

i,

aux = fila \* dimens.columnas;

if ( (fila >= dimens.filas)

|| ( (blockIdx.x \* blockDim.x + threadIdx.x) != 0) )

{

return;

}

/\* Recorre la matriz marcando los elementos iguales consecutivos \*/

for (i = 0; i < (dimens.columnas - 2) ; i++)

{

if ( (matriz [aux + i] == matriz [aux + i + 1])

&& (matriz [aux + i] == matriz [aux + i + 2]) )

{

coincidencias [aux + i] = COINCIDE;

coincidencias [aux + i + 1] = COINCIDE;

coincidencias [aux + i + 2] = COINCIDE;

}

}

}

/\*\*

\* Comprueba si la columna contiene elementos repetidos.

\*

\* @param matriz

\* Matriz con los valores actuales de los diamantes.

\*

\* @param dimens

\* Estructura con las dimensiones de la matriz.

\*

\* @param coincidencias

\* Matriz en la que se va a indicar si había alguna coincidencia.

\*/

\_\_global\_\_ void busar\_coinc\_cuda\_col (int \*matriz,

Dim dimens,

int \*coincidencias)

{

int columna = blockIdx.x \* blockDim.x + threadIdx.x,

i;

if ( (columna >= dimens.columnas)

|| ( (blockIdx.y \* blockDim.y + threadIdx.y) != 0) )

{

return;

}

/\* Recorre la matriz marcando los elementos iguales consecutivos \*/

for (i = 0; i < (dimens.filas - 2) ; i++)

{

if ( (matriz [(i \* dimens.columnas) + columna]

== matriz [( (i + 1) \* dimens.columnas) + columna])

&& (matriz [(i \* dimens.columnas) + columna]

== matriz [( (i + 2) \* dimens.columnas) + columna]) )

{

coincidencias [(i \* dimens.columnas) + columna] = COINCIDE;

coincidencias [(i + 1) \* dimens.columnas + columna] = COINCIDE;

coincidencias [(i + 2) \* dimens.columnas + columna] = COINCIDE;

}

}

}

/\*\*

\* Elimina todos los elementos que se haya visto que han coincidido.

\*

\*

\* @param matriz

\* Matriz con los valores actuales de los diamantes.

\*

\* @param dimens

\* Estructura con las dimensiones de la matriz.

\*

\* @param coincidencias

\* Matriz con las coincidencias encontradas.

\*/

\_\_global\_\_ void eliminar\_coinc\_cuda (int \*matriz,

Dim dimens,

int \*coincidencias)

{

int fila = blockIdx.y \* blockDim.y + threadIdx.y,

columna = blockIdx.x \* blockDim.x + threadIdx.x;

if ( (fila >= dimens.filas)

|| (columna >= dimens.columnas))

{

return;

}

if (coincidencias [(fila \* dimens.columnas) + columna] == COINCIDE)

{

matriz [(fila \* dimens.columnas) + columna] = DIAMANTE\_VACIO;

}

}

/\*\*

\* Comprueba todos los huecos de la columna y rellena los vacíos.

\*

\* @param semilla

\* Elemento inicial para generar la secuencia.

\*

\* @param resultado

\* Vector en el que se almacenarán los números generados.

\*

\* @param min

\* Límite inferior para generar un número (inclusivo).

\*

\* @param max

\* Límite superior para generar un número (inclusivo).

\*

\* @param dimens

\* Dimensiones de la matriz resultado.

\*/

\_\_global\_\_ void llenar\_vacios\_cuda (unsigned long semilla,

int \*resultado,

const int min,

const int max,

const Dim dimens)

{

int columna = blockIdx.x \* blockDim.x + threadIdx.x,

i,

elem,

rand\_int;

curandState estado;

float rand\_f;

if ( (columna >= dimens.columnas)

|| ( (blockIdx.y \* blockDim.y + threadIdx.y) != 0) )

{

return;

}

/\* Recorre la columna hasta encontrar un elemento vacío \*/

for (i = (dimens.filas - 1); i >= 0; i--)

{

elem = resultado [(i \* dimens.columnas) + columna];

if (elem == DIAMANTE\_VACIO)

{

/\* Busca el primer elemento que haya por encima y lo baja \*/

elem = buscar\_lleno (resultado, i, columna, dimens);

if (elem == -1)

{

curand\_init (semilla, i + columna, 0, &estado);

/\* El número se genera primero con coma flotante

(ajustando los límites como se haya especificado) y

luego se convierte a entero. Esto es más rápido que

realizar la operación de módulo \*/

rand\_f = curand\_uniform (&estado);

rand\_f \*= (max - min + 0.999999);

rand\_f += min;

/\* Convierte el float a entero \*/

rand\_int = \_\_float2int\_rz (rand\_f);

/\* Guarda el resultado \*/

elem = rand\_int;

}

resultado [(i \* dimens.columnas) + columna] = elem;

}

}

}

/\* -------------------- \*/

/\* FUNCIONES AUXILIARES \*/

/\* -------------------- \*/

/\*\*

\* Obtiene las dimensiones de los hilos necesarias para ejecutar la matriz con las

\* dimensiones especificadas, teniendo en cuenta las limitaciones del dispositivo.

\*

\* @param bloques

\* Elemento de tipo dim3 para almacenar las dimensiones de los bloques

\* dentro de la rejilla (2D).

\*

\* @param hilos

\* Elemento de tipo dim3 para almacenar las dimensiones de los hilos dentro

\* de los bloques (3D).

\*

\* @param tam\_matriz

\* Estructura Dim (definida en 'commno.h') con las dimensiones de la matriz

\* que se desea usar en el dispositivo.

\*

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int obtener\_dim (dim3 \*bloques, dim3 \*hilos, Dim tam\_matriz)

{

cudaDeviceProp propiedades;

cudaError\_t err;

int id\_dispos = -1;

/\* Busca el dispositivo con versión >= 2 (para poder usar más hilos

por bloque) \*/

propiedades.major = 2;

CUDA (err,

cudaChooseDevice (&id\_dispos, &propiedades)

);

/\* Actualiza la información del dispositivo (chooseDevice no lo hizo

correctamente) \*/

CUDA (err,

cudaGetDeviceProperties (&propiedades, id\_dispos)

);

imprimir (DETALLE\_EXTRA, "\n -> Escogido dispositivo %d, con versión %d.%d\n\n",

id\_dispos,

propiedades.major, propiedades.minor);

cudaSetDevice (id\_dispos);

/\* Número constante de hilos por bloque (para versiones anteriores

a Fermi, 16 hilos) \*/

hilos->x = (propiedades.major < 2)? 16 : 32;

hilos->y = (propiedades.major < 2)? 16 : 32;

hilos->z = 1;

/\* Se calcula el número de bloques que se deben utilizar \*/

bloques->x = ceil (((float) tam\_matriz.columnas) / ((float) hilos->x));

bloques->y = ceil (((float) tam\_matriz.filas) / ((float) hilos->y));

bloques->z = 1;

imprimir (DETALLE\_EXTRA, "Se usan bloques de %d x %d para alojar los (%d x %d)"

" hilos necesarios.\n",

hilos->x, hilos->y,

tam\_matriz.columnas, tam\_matriz.filas);

/\* Si la matriz no cabe, se avisa \*/

if ((bloques->x > propiedades.maxGridSize [0])

|| (bloques->y > propiedades.maxGridSize [1]))

{

imprimir (DETALLE\_LOG, "\n -> Error: la matriz es demasiado grande "

"para el dispositivo\n");

return ERR\_TAM;

}

return SUCCESS;

}

/\*\*

\* Copia la información de la matriz de enteros (el resultado devuelto por el

\* dispositivo) en la matriz de juego con la que trabaja la CPU.

\*

\* @param matriz\_d

\* Matriz de enteros con los resultados de la tarjeta.

\*

\* @param malla

\* Malla con la información del juego, cuya matriz va a ser actualizada.

\*/

void copiar\_matriz (int \*matriz\_d, Malla \*malla)

{

int i,

j,

idx,

filas = malla->dimens.filas,

columnas = malla->dimens.columnas;

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

{

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

{

idx = (i \* columnas) + j;

malla->matriz [idx] = crear\_diamante (matriz\_d [idx]);

}

}

}

/\*\*

\* Rellena la matriz de juego con diamantes aleatorios.

\*

\* @param malla

\* Estructura de tipo Malla (definida en 'common.h') con las dimensiones de

\* la matriz y su contenido.

\*

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si alguna función CUDA ha fallado.

\*/

int matriz\_aleat (Malla \*malla)

{

int max = DIAMANTE\_VACIO,

filas = malla->dimens.filas,

columnas = malla->dimens.columnas,

tam = filas \* columnas;

cudaError\_t err;

dim3 bloques,

hilos;

int \*matriz\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

max = max\_nv (\*malla);

/\* Comprueba que la matriz tiene memoria reservada \*/

if (malla->matriz == NULL)

{

imprimir (DETALLE\_DEBUG,

"Error: la matriz no tiene memoria reservada.\n");

return ERR\_MEM;

}

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d, tam \* sizeof matriz\_d [0])

);

/\* Llama al núcleo para inicializar la secuencia de números aleatorios \*/

obtener\_dim (&bloques, &hilos, malla->dimens);

/\* Genera los números aleatorios y los copia en la matriz \*/

KERNEL (err, gen\_aleat\_cuda,

bloques, hilos,

time (NULL), matriz\_d, 1, max, malla->dimens

);

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

/\* Se libera la memoria del dispositivo \*/

CUDA (err,

cudaFree (matriz\_d)

);

return SUCCESS;

}

/\*\*

\* Función para ejecutar la bomba I (eliminar fila).

\*

\* @param fila\_bomba

\* Fila que se debe eliminar (poner a DIAMANTE\_VACIO).

\*

\* @param malla

\* Estructura con la información del juego.

\*

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int bomba\_fila (int fila\_bomba, Malla \*malla)

{

cudaError\_t err;

dim3 bloques,

hilos;

int tam = malla->dimens.filas \* malla->dimens.columnas,

i,

j,

idx = 0,

max = max\_nv (\*malla);

int \*matriz\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

/\* Dimensiones para luego crear un hilo por columna \*/

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = 1;

dim\_matr\_hilos.columnas = malla->dimens.columnas;

/\* Inicializa la matriz auxiliar \*/

for (i = 0; i < malla->dimens.filas; i++)

{

for (j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux [idx] = malla->matriz [idx].id;

}

}

/\* Reserva memoria en el dispositivo y copia la matriz \*/

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d,

tam \* sizeof matriz\_d [0])

);

CUDA (err,

cudaMemcpy (matriz\_d, aux, tam \* sizeof matriz\_d [0],

cudaMemcpyHostToDevice)

);

/\* Llama al núcleo para eliminar la fila \*/

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, eliminar\_fila\_cuda,

bloques, hilos,

time (NULL), matriz\_d, 1, max, malla->dimens, fila\_bomba

);

/\* Copia la información de vuelta y libera la memoria en el dispositivo \*/

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

CUDA (err,

cudaFree (matriz\_d)

);

return SUCCESS;

}

/\*\*

\* Función para ejecutar la bomba II (eliminar columna).

\*

\* @param col\_bomba

\* Columna que se debe eliminar (poner a DIAMANTE\_VACIO).

\*

\* @param malla

\* Estructura con la información del juego.

\*

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int bomba\_columna (int col\_bomba, Malla \*malla)

{

cudaError\_t err;

dim3 bloques,

hilos;

int tam = malla->dimens.filas \* malla->dimens.columnas,

i,

j,

idx = 0,

max = max\_nv (\*malla);

int \*matriz\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

/\* Dimensiones para luego crear un hilo por columna \*/

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = malla->dimens.filas;

dim\_matr\_hilos.columnas = 1;

/\* Inicializa la matriz auxiliar \*/

for (i = 0; i < malla->dimens.filas; i++)

{

for (j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux [idx] = malla->matriz [idx].id;

}

}

/\* Reserva memoria en el dispositivo y copia la matriz \*/

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d,

tam \* sizeof matriz\_d [0])

);

CUDA (err,

cudaMemcpy (matriz\_d, aux, tam \* sizeof matriz\_d [0],

cudaMemcpyHostToDevice)

);

/\* Llama al núcleo para eliminar la columna \*/

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, eliminar\_columna\_cuda,

bloques, hilos,

time (NULL), matriz\_d, 1, max, malla->dimens, col\_bomba

);

/\* Copia la información de vuelta y libera la memoria en el dispositivo \*/

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

CUDA (err,

cudaFree (matriz\_d)

);

return SUCCESS;

}

/\*\*

\* Función para ejecutar la bomba III (girar en grupos de 3x3).

\*

\* @param malla

\* Estructura con toda la información del juego (matriz, nivel

\* y dimensiones).

\*

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int bomba\_giro (Malla \*malla)

{

cudaError\_t err;

dim3 bloques,

hilos;

int tam = malla->dimens.filas \* malla->dimens.columnas,

i,

j,

idx = 0;

int \*matriz\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

/\* Inicializa la matriz auxiliar \*/

for (i = 0; i < malla->dimens.filas; i++)

{

for (j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux [idx] = malla->matriz [idx].id;

}

}

/\* Reserva memoria en el dispositivo y copia la matriz \*/

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d,

tam \* sizeof matriz\_d [0])

);

CUDA (err,

cudaMemcpy (matriz\_d, aux, tam \* sizeof matriz\_d [0],

cudaMemcpyHostToDevice)

);

/\* Llama al núcleo para girar la matriz \*/

obtener\_dim (&bloques, &hilos, malla->dimens);

KERNEL (err, girar\_matriz\_cuda,

bloques, hilos,

matriz\_d, malla->dimens

);

/\* Copia la información de vuelta y libera la memoria en el dispositivo \*/

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

CUDA (err,

cudaFree (matriz\_d)

);

return SUCCESS;

}

/\*\*

\* Busca coincidencias en la matriz y marca las casillas para ser eliminadas (las deja

\* como DIAMANTE\_VACIO.

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int eliminar\_coincidencias (Malla \*malla)

{

cudaError\_t err;

dim3 bloques,

hilos;

int tam = malla->dimens.filas \* malla->dimens.columnas,

i,

j,

idx = 0;

int \*matriz\_d,

\*coincidencias\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

/\* Dimensiones para luego crear un hilo por columna \*/

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = malla->dimens.filas;

dim\_matr\_hilos.columnas = 1;

/\* Inicializa la matriz auxiliar \*/

for (i = 0; i < malla->dimens.filas; i++)

{

for (j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux [idx] = malla->matriz [idx].id;

}

}

/\* Reserva memoria en el dispositivo y copia la matriz \*/

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d,

tam \* sizeof matriz\_d [0])

);

CUDA (err,

cudaMalloc ((void \*\*) &coincidencias\_d,

tam \* sizeof coincidencias\_d [0])

);

CUDA (err,

cudaMemset (coincidencias\_d, NO\_COINCIDE,

tam \* sizeof coincidencias\_d [0])

);

CUDA (err,

cudaMemcpy (matriz\_d, aux, tam \* sizeof matriz\_d [0],

cudaMemcpyHostToDevice)

);

/\* Llama al núcleo para comprobar la matriz \*/

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, busar\_coinc\_cuda\_fila,

bloques, hilos,

matriz\_d, malla->dimens, coincidencias\_d

);

dim\_matr\_hilos.filas = 1;

dim\_matr\_hilos.columnas = malla->dimens.columnas;

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, busar\_coinc\_cuda\_col,

bloques, hilos,

matriz\_d, malla->dimens, coincidencias\_d

);

/\* Utiliza la matriz con los elementos marcados para eliminarlos \*/

obtener\_dim (&bloques, &hilos, malla->dimens);

KERNEL (err, eliminar\_coinc\_cuda,

bloques, hilos,

matriz\_d, malla->dimens, coincidencias\_d

);

/\* Copia la información de vuelta y libera la memoria en el dispositivo \*/

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

CUDA (err,

cudaFree (matriz\_d)

);

CUDA (err,

cudaFree (coincidencias\_d)

);

return SUCCESS;

}

/\*\*

\* Rellena los diamantes vacíos en la matriz.

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo algún error al obtener las características del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int llenar\_vacios (Malla \*malla)

{

cudaError\_t err;

dim3 bloques,

hilos;

int tam = malla->dimens.filas \* malla->dimens.columnas,

i,

j,

idx = 0,

max = max\_nv (\*malla);

int \*matriz\_d,

\*aux = (int \*) malloc (tam \* sizeof aux [0]);

/\* Dimensiones para luego crear un hilo por columna \*/

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = 1;

dim\_matr\_hilos.columnas = malla->dimens.columnas;

/\* Inicializa la matriz auxiliar \*/

for (i = 0; i < malla->dimens.filas; i++)

{

for (j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux [idx] = malla->matriz [idx].id;

}

}

/\* Reserva memoria en el dispositivo y copia la matriz \*/

CUDA (err,

cudaMalloc ((void \*\*) &matriz\_d,

tam \* sizeof matriz\_d [0])

);

CUDA (err,

cudaMemcpy (matriz\_d, aux, tam \* sizeof matriz\_d [0],

cudaMemcpyHostToDevice)

);

/\* Llama al núcleo para comprobar la matriz \*/

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, llenar\_vacios\_cuda,

bloques, hilos,

time (NULL), matriz\_d, 1, max, malla->dimens

);

/\* Copia la información de vuelta y libera la memoria en el dispositivo \*/

CUDA (err,

cudaMemcpy (aux, matriz\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost)

);

/\* Copiar directamente un array de Diamante desde el dispositivo da problemas,

así que se usa un array de enteros para crear los números aleatorios en

paralelo y luego la CPU se encarga de crear los elementos de tipo Diamante \*/

copiar\_matriz (aux, malla);

CUDA (err,

cudaFree (matriz\_d)

);

return SUCCESS;

}

/\*\*

\* Busca las mejores jugadas por filas

\*

\* @param matriz

\* Matriz con los valores actuales de los diamantes.

\*

\* @param dimens

\* Estructura con las dimensiones de la matriz.

\*

\* @param mat2

\* Matriz en la que se va a indicar los valores de las mejores

\* jugadas por posicion

\*

\* @param solh

\* Matriz que devuleve las mejores jugdas(posicion, movimiento

\* y valor de cada jugada).

\*/

\_\_global\_\_ void realizar\_jugada\_horizontal\_cuda (int \* mat1, Dim dimens, int \* mat2,int \* solh)

{

int fila = blockIdx.x \* blockDim.x + threadIdx.x;

int aux = fila \* dimens.columnas;

if ((fila >= dimens.filas)

|| ( (blockIdx.y \* blockDim.y + threadIdx.y) != 0) )

{

return;

}

for(int i = 0; i < dimens.filas \* 4; i++){

solh[i] = 0;

}

//Variable para recorrer la matriz

int fin = 0;

int cont = 0;

int mov = 1;

int ref = 0;

//Vareables no definitivas para guardar el movimiento que se va a realizar

int sen = 0;

int posY = 0;

int posX = 0;

//Mejor movimiento horizontal

int mh\_valor = 0;

int mh\_sen = 0; //Sentido del movimeinto

int mh\_posY = 0;

int mh\_posX = 0;

for (int i = 0; i < dimens.columnas; i++)

{

ref = mat1[aux + i];

for(int j = i; j < dimens.columnas; j ++)

{

if(fin == 0)

{

if(ref == mat1[aux + j])

{

//Mira si la posicion en la que esta es igual a la referencia

cont ++;

}

else if((mov == 1)&&(fila > 0)&&(ref == mat1[aux - dimens.columnas + j]))

{

//Mira la posicion de arriba --> mover mat1[aux + j] arriba

mov = 0;

cont ++;

sen = 3; posY = fila; posX = j;

}

else if((mov == 1)&&(fila < dimens.filas - 1)&&(ref == mat1[aux + dimens.columnas + j]))

{

//Mira la posicion de abajo --> mover mat1[aux + j] abajo

mov = 0;

cont ++;

sen = 1; posY = fila;posX = j;

}

else if((mov == 1)&&((j + 1) < dimens.columnas)&&(ref == mat1[aux + j + 1]))

{

//Mirar la posicion de la derecha --> mover mat1[aux + j] derecha

mov = 0;

cont ++;

sen = 0; posY = fila; posX = j;

j++; //Pasa a comprobar la siguiente

}

else

{

fin = 1;

}

}

}

//Mirar en las posiciones de la izquierda

if ((mov == 1)&&(i > 0)&&(ref == mat1 [aux - dimens.columnas + i - 1]))

{

//Mirar la posicion por el lado de la izquierda arriba --> mover mat[aux + i - 1] arriba

mov = 0;

cont ++;

sen = 3; posY = fila; posX = i - 1;

}

else if ((mov == 1)&&(i > 0)&&(ref == mat1 [aux + dimens.columnas + i -1]))

{

//Mirar la posicion por el lado de la izquierda abajo --> mover mat[aux + i - 1] abajo

mov = 0;

cont ++;

sen = 1; posY = fila; posX = i - 1;

}

if(solh[fila \* 4] <= cont){

mh\_sen = sen;

if (mov == 1) sen = 4;

mh\_posY = posY;

mh\_posX = posX;

}

if((solh[fila \* 4] == 0)||(solh [fila \* 4] < cont))

{

solh[fila \* 4] = cont;

solh[(fila \* 4) + 1] = mh\_sen;

solh[(fila \* 4) + 2] = mh\_posY;

solh[(fila \* 4) + 3] = mh\_posX;

}

mat2[aux + i] = cont;

//Reinicia valores

mov = 1;

fin = 0;

cont = 0;

}

}

/\*\*

\* Busca las mejores jugadas por filas

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo alg˙n error al obtener las caracterÌsticas del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int realizar\_jugada\_horizontal(Malla \* malla,int \* jugada)

{

cudaError\_t err;

dim3 bloques, hilos;

int tam = malla->dimens.columnas \* malla->dimens.filas;

int \* matriz\_d;

int \* mov\_d;

int \* aux = (int \*) malloc (tam \* sizeof aux [0]);

//vector solh

int \* vec = (int \*) malloc (malla->dimens.filas \* sizeof (int) \* 4);

int \* vec\_d;

cudaMalloc((void \*\*) &vec\_d,malla->dimens.filas \* sizeof (int) \* 4);

for (int i = 0; i < malla->dimens.filas \* 4; i++)

{

vec[i] = 0;

}

//Crea un hilo por columna

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = malla->dimens.filas;

dim\_matr\_hilos.columnas = 1;

//Inicializa la matriz auxiliar

int idx;

for (int i = 0; i < malla->dimens.filas; i++)

{

for (int j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux[idx] = malla->matriz[idx].id;

}

}

CUDA (err, cudaMalloc((void \*\*)&matriz\_d,tam \* sizeof matriz\_d[0]));

CUDA (err, cudaMalloc((void \*\*)&mov\_d,tam \* sizeof mov\_d[0]));

CUDA (err, cudaMemset(mov\_d, NO\_COINCIDE, tam \* sizeof mov\_d[0]));

CUDA (err, cudaMemcpy(matriz\_d, aux, tam \* sizeof matriz\_d [0],cudaMemcpyHostToDevice));

cudaMemcpy (vec\_d, &vec , sizeof (int) \* malla->dimens.filas , cudaMemcpyHostToDevice);

/\* Llama al nucelo para comprar la matriz \*/

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, realizar\_jugada\_horizontal\_cuda, bloques,hilos,matriz\_d,malla->dimens,mov\_d,vec\_d);

CUDA (err, cudaMemcpy(aux,mov\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost));

CUDA (err, cudaMemcpy(vec, vec\_d, sizeof(int)\*malla->dimens.filas \*4,cudaMemcpyDeviceToHost));

for (int i = 0; i < malla->dimens.filas; i++)

{

if(jugada[0]<vec[i \* 4])

{

jugada[0] = vec[i \* 4];

jugada[1] = vec[i \* 4 +1];

jugada[2] = vec[i \* 4 +2];

jugada[3] = vec[i \* 4 +3];

}

}

//copiar\_matriz (aux, malla);

CUDA (err, cudaFree(matriz\_d));

CUDA (err, cudaFree(mov\_d));

return SUCCESS;

}

/\*\*

\* Busca las mejores jugadas por columnas

\*

\* @param matriz

\* Matriz con los valores actuales de los diamantes.

\*

\* @param dimens

\* Estructura con las dimensiones de la matriz.

\*

\* @param mat2

\* Matriz en la que se va a indicar los valores de las mejores

\* jugadas por posicion

\*

\* @param solv

\* Matriz que devuleve las mejores jugdas(posicion, movimiento

\* y valor de cada jugada).

\*/

\_\_global\_\_ void realizar\_jugada\_vertical\_cuda (int \* mat1, Dim dimens, int \* mat2, int \* solv)

{

int columna = blockIdx.y \* blockDim.y + threadIdx.y;

if ((columna >= dimens.columnas)

|| ( (blockIdx.x \* blockDim.x + threadIdx.x) != 0) )

{

return;

}

for(int i = 0;i < dimens.columnas \* 4; i++){

solv[i] = 0;

}

//Variable para recorrer la matriz

int fin = 0;

int cont = 0;

int mov = 1;

int ref = 0;

//Vareables no definitivas para guardar el movimiento que se va a realizar

int sen = 0;

int posY = 0;

int posX = 0;

//Mejor movimiento horizontal

int mh\_valor = 0;

int mh\_sen = 0; //Sentido del movimeinto

int mh\_posY = 0;

int mh\_posX = 0;

for (int i = 0; i < dimens.filas; i++)

{

ref = mat1[dimens.columnas \* i + columna];

for(int j = i; j < dimens.filas; j ++)

{

if(fin == 0)

{

if(ref == mat1[dimens.columnas \* j + columna])

{

//Mira si la posicion en la que esta es igual a la referencia

cont ++;

}

else if((mov == 1)&&(columna > 0)&&(ref == mat1[dimens.columnas \* j + columna - 1]))

{

//Mira la posicion de la izquierda --> mover mat1[dimens.columnas \* j + columna - 1] izquierda

mov = 0;

cont ++;

sen = 2; posY = j; posX = columna;

}

else if((mov == 1)&&(columna < dimens.columnas - 1)&&(ref == mat1[dimens.columnas \* j + columna + 1]))

{

//Mira la posicion de la derecha --> mover mat1[dimens.columnas \* j + columna + 1] derecha

mov = 0;

cont ++;

sen = 0; posY = j;posX = columna;

}

else if((mov == 1)&&((j + 1) < dimens.filas)&&(ref == mat1[dimens.columnas \* (j+1) + columna]))

{

//Mirar la posicion de abajo --> mover mat1[dimens.columnas \* j + columna] abajo

mov = 0;

cont ++;

sen = 1; posY = j; posX = columna;

j++; //Pasa a comprobar la siguiente

}

else

{

fin = 1;

}

}

}

//Mirar en las posiciones de arriba

if ((mov == 1)&&(columna > 0)&&(i>0)&&(ref == mat1 [dimens.columnas \* (i- 1) + (columna - 1)]))

{

//Mirar la posicion por el lado de arriba izquierda --> mover mat[dimens.columnas \* (i- 1) + (columna - 1)] izquierda

mov = 0;

cont ++;

sen = 2; posY = i -1 ; posX =columna;

}

else if ((mov == 1)&&(i > 0)&&(columna + 1 < dimens.columnas)&&(ref == mat1 [dimens.columnas \* (i - 1) + (columna + 1)]))

{

//Mirar la posicion por el lado de la arriba derecha --> mover mat[dimens.columnas \* (i - 1) + (columna + 1)] derecha

mov = 0;

cont ++;

sen = 0; posY = i - 1; posX = columna;

}

if(solv[columna \* 4] <= cont){

mh\_sen = sen;

if (mov == 1) sen = 4;

mh\_posY = posY;

mh\_posX = posX;

}

if((solv[columna \* 4] == 0)||(solv[columna \* 4] < cont))

{

solv[columna \* 4] = cont;

solv[(columna \* 4) + 1] = mh\_sen;

solv[(columna \* 4) + 2] = mh\_posY;

solv[(columna \* 4) + 3] = mh\_posX;

}

mat2[dimens.columnas \* i + columna] = cont;

//Reinicias valores

mov = 1;

fin = 0;

cont = 0;

}

}

/\*\*

\* Busca las mejores jugadas por columnas

\*

\* @return

\* SUCCESS si todo ha salido correctamente.

\* ERR\_CUDA si hubo alg˙n error al obtener las caracterÌsticas del

\* dispositivo.

\* ERR\_TAM si la matriz especificada sobrepasa las capacidades del

\* dispositivo.

\*/

int realizar\_jugada\_vertical(Malla \* malla,int \* jugada)

{

cudaError\_t err;

dim3 bloques, hilos;

int tam = malla->dimens.columnas \* malla->dimens.filas;

int \* matriz\_d;

int \* mov\_d;

int \* aux = (int \*) malloc (tam \* sizeof aux [0]);

int \* vec = (int \*) malloc (malla->dimens.columnas \* sizeof (int) \* 4);

int \* vec\_d;

cudaMalloc((void \*\*) &vec\_d,malla->dimens.columnas \* sizeof (int) \* 4);

for (int i = 0; i < malla->dimens.columnas \* 4; i++)

{

vec[i] = 0;

}

//Crea un hilo por fila

Dim dim\_matr\_hilos;

dim\_matr\_hilos.filas = 1;

dim\_matr\_hilos.columnas = malla->dimens.columnas;

//Inicializa la matriz auxiliar

int idx;

for (int i = 0; i < malla->dimens.filas; i++)

{

for (int j = 0; j < malla->dimens.columnas; j++)

{

idx = (i \* malla->dimens.columnas) + j;

aux[idx] = malla->matriz[idx].id;

}

}

CUDA (err, cudaMalloc((void \*\*)&matriz\_d,tam \* sizeof matriz\_d[0]));

CUDA (err, cudaMalloc((void \*\*)&mov\_d,tam \* sizeof mov\_d[0]));

CUDA (err, cudaMemset(mov\_d, NO\_COINCIDE, tam \* sizeof mov\_d[0]));

CUDA (err, cudaMemcpy(matriz\_d, aux, tam \* sizeof matriz\_d [0],cudaMemcpyHostToDevice));

cudaMemcpy( vec\_d, &vec,sizeof (int) \* malla->dimens.columnas, cudaMemcpyHostToDevice );

obtener\_dim (&bloques, &hilos, dim\_matr\_hilos);

KERNEL (err, realizar\_jugada\_vertical\_cuda, bloques,hilos,matriz\_d,malla->dimens,mov\_d,vec\_d);

CUDA (err, cudaMemcpy(aux,mov\_d, tam \* sizeof aux [0], cudaMemcpyDeviceToHost));

CUDA (err, cudaMemcpy(vec, vec\_d,sizeof (int) \* malla->dimens.columnas \* 4, cudaMemcpyDeviceToHost ));

for (int i = 0; i < malla->dimens.columnas; i++)

{

if(jugada[0]<vec[i \* 4])

{

jugada[0] = vec[i \* 4];

jugada[1] = vec[i \* 4 +1];

jugada[2] = vec[i \* 4 +2];

jugada[3] = vec[i \* 4 +3];

}

}

CUDA (err, cudaFree(matriz\_d));

CUDA (err, cudaFree(mov\_d));

return SUCCESS;

}

int realizar\_jugada(Malla \* malla)

{

int jugada\_v[4];

int jugada\_h[4];

int posY;

int posX;

int mov;

realizar\_jugada\_vertical(malla,jugada\_v);

realizar\_jugada\_horizontal(malla,jugada\_h);

if(jugada\_v[0]>jugada\_h[0])

{

printf("Mejor jugada --> Mov(%d): %d PosY: %d PosX: %d\n",jugada\_v[0],jugada\_v[1],jugada\_v[2],jugada\_v[3]);

posY = jugada\_v[2];

posX = jugada\_v[3];

mov = jugada\_v[1];

}

else

{

printf("Mov(%d): %d PosY: %d PosX: %d\n",jugada\_h[0],jugada\_h[1],jugada\_h[2],jugada\_h[3]);

posY = jugada\_h[2];

posX = jugada\_h[3];

mov = jugada\_h[1];

}

mover\_diamante(posY,posX,mov,\* malla);

return SUCCESS;

}