CUNDIO CRACK SAGA

Desarrollado en Cuda

Eduardo García Huerta y Jaime Díez Buendía

Índice

Implementación básica

Características clave

Cundio Crack saga es un programa realizado mediante el lenguaje CUDA que simula un funcionamiento similar al famoso juego móvil Candy Crush Saga.
En nuestra implementación se dispone del "device" gracias al que se podrán hacer llamadas con varios hilos, bloques y una memoria compartida que a la que tendrán acceso los distintos bloques, de manera que se podrá realizar una programación concurrente explotando al máximo la tarjeta gráfica Nvidia que presenta el ordenador.

La implementación básica funcionará con NxM hilos y 1 solo bloque y esta constará de lo siguiente:

Una matriz que supondrá un tablero finito de dimensiones NxM, que serán introducidas por el usuario.
Esta matriz representa el tablero en el que el usuario podrá jugar.
Dentro de este "tablero" habrá distintos "caramelos", que se representarán mediante los números del 1 al 6, adicionalmente, cada uno de ellos se muestra de un color distinto, para ser más fieles al juego y que además facilite el juego al usuario. Estos colores son azul, rojo, naranja, verde, marrón y lila respectivamente.
De esta manera, gracias a la función de imprimir la matriz se vería de la siguiente manera:

_	0	1	2	3	4	5	6	7	8
0	6	5	4	4	4	1	1	6	1
1	1	5	4	3	3	4	6	3	5
2	3	5	3	6	3	6	2	5	5
3	1	2	4	5	2	6	2	3	4
4	4	2	1	4	4	3	4	2	6
5	2	6	6	3	3	1	2	6	2
6	6	3	6	4	1	1	4	2	3
7	1	4	1	4	5	5	3	1	1
8	6	3	3	4	2	4	4	2	5

Imagen 1.1

A su vez también habrá distintos "caramelos" especiales que tienen sus propias características. Estos son:

- **La bomba:** se representará con una "B" para el usuario, no obstante, para evitar problemas con el código, esta esta referenciada como un 7 ya que la matriz que se ha creado es una matriz de integers. Si se explota este "caramelo" especial, explotará de forma aleatoria toda una fila o toda una columna. Este caramelo aparecerá en el momento que escojas un caramelo

a explotar y exploten exactamente 5 caramelos (el caramelo seleccionado y 4 adyacencias a este)

- **La TNT:** esta se representará al usuario mediante una “T”, no obstante, entro del código se representará como un 8. Este se obtiene al romper exactamente 6 caramelos, y su función especial es que explotan todos los caramelos de su alrededor en un radio de 4*4 filas y columnas.
- **El rompecabezas:** este se representará como “Rx” donde la x es un numero entre 1 y 6. Este se obtiene cuando rompes 7 o más caramelos. En ese momento, se formará un rompecabezas aleatorio (R1, R2, R3, R4, R5 o R6). Cuando este explota, todos los números que corresponden a la X serán borrados del tablero, por ejemplo, si explotamos un R1, todos los 1 que estén en el tablero explotarán.

Por otra parte, para dejar más limpio el código, hemos decidido de en vez de hacer todas las llamadas al kernel en el main, hemos decido crear funciones el host, donde en ellas se inicializan todos los parámetros necesarios para llamar al kernel. Un ejemplo de esto sería el siguiente.

```
void crear_matriz_aleatoria(int* mat, int n, int m, int lim_inf, int lim_sup) {
    //creacion de un puntero para la GPU
    int* d_mat;
    //reservamos espacio en la memoria
    cudaMalloc((void**)&d_mat, n * m * sizeof(int));

    //puntero que ayudara a la obtencion de numeros aleatorios dentro del Kernel
    curandState* d_state;
    //reservamos espacio en la memoria
    cudaMalloc((void**)&d_state, n * m * sizeof(curandState));

    //obtencion de una semilla que ayudara a la creacion de numeros aleatorios
    unsigned int ale = generate_seed();

    dim3 block_size(m,n);
    dim3 num_blocks(1,1);
    matriz_aleatoria <<<num_blocks, block_size >> > (d_mat, n, m, lim_inf, lim_sup, ale, d_state);

    //Copiamos en el host el resultado obtenido en el device
    cudaMemcpy(mat, d_mat, n * m * sizeof(int), cudaMemcpyDeviceToHost);

    //liberacion de punteros
    cudaFree(d_mat);
    cudaFree(d_state);
}
```

Imagen 1.2

Como podemos observar en la imagen, la función crear_matriz_aleatoria recibe los parámetros que se necesitan para poder hacer la llamada al kernel “matriz aleatoria”. Dentro de la función se inicializan los valores para poder ejecutar el kernel. Una vez que se haya completado la función de

la GPU, (mostrada en una imagen a continuación), debemos retornar de vuelta los elementos obtenidos en la GPU. Tras todo esto se liberarán los punteros creados para no malgastar memoria.

```
// Esta función genera una matriz aleatoria de números enteros entre "lim_inf" y "lim_sup".
__global__ void matriz_aleatoria(int* mat, int n, int m, int lim_inf, int lim_sup, unsigned int ale, curandState* state) {
    // Calcular las coordenadas x e y del hilo
    int idx = threadIdx.x;
    int idy = threadIdx.y;

    // Verificar si el hilo se encuentra dentro de los límites de la matriz
    if (idx < m && idy < n) {
        // Inicializar el generador de números aleatorios
        curand_init(ale, idy * m + idx, 0, &state[idy * m + idx]);
        // Generar un número aleatorio entero entre "lim_inf" y "lim_sup"
        int val = curand(&state[idy * m + idx]) % lim_sup + lim_inf;
        // Asignar el valor aleatorio a la matriz
        mat[idy * m + idx] = val;
    }
}
```

Imagen 1.3

Con esta función podemos observar cómo obtenemos las posiciones de los hilos, done el idx corresponde a la columna que le corresponde al hilo dentro de la matriz, y el idy es la fila que le correspondería a este. La posición exacta que le corresponderá al hilo se calcula mediante la siguiente fórmula: posición=idy*m (número de columnas) +idx. De esta manera, cada hilo calcula un número aleatorio y lo asignará a la posición correspondiente en la matriz, concluyendo así la creación de la matriz aleatoria.

Todas las funciones tienen la misma estructura, pero con respectivo código para cada objetivo, pero con la intención de hacer esta memoria lo más breve posible, mostraremos los ejemplos anteriores y algunas otras funciones más reseñables. Para más información sobre las funciones se adjuntan los códigos de los programas.

Otra cosa que debemos aclarar es que, según el enunciado, comprendemos que todos los métodos que se deben ejecutarse en la GPU son todos aquellos que modifiquen la matriz. De esta forma todas aquellas funciones que no modifiquen estrictamente la matriz las haremos como funciones auxiliares en el host cómo, por ejemplo, la función de buscar las adyacencias.

Esta función (llamada ver_candy) es importante en nuestro programa y, gracias a ella obtendremos las posiciones adyacentes al número que hayamos seleccionado cuyos elementos sean iguales que nuestro número y, se guardarán en un vector que se utilizará posteriormente como un elemento auxiliar para la eliminación de los caramelos.
Lo utilizaremos de la siguiente manera: se lo pasamos a los kernels, que comprueban si su hilo es igual a algún elemento del vector, en cuyo caso, eliminarán este elemento del vector y su posición de la matriz también será reemplazada por -1 (que es el número que usamos para representar elementos borrados).

```
void ver_candy(int* mat, int n, int m,int colum,int fila, int* vector,int elemento) {
    int caramelo = fila * m + colum;//posicion en la matriz de las coordenadas

    if (!esta_en_vector(vector,m,n,caramelo)&& mat[caramelo]==elemento) {
        //comprobamos que la posicion no ha sido ya insertada
        int pos=primer_vacio(vector,n,m);
        vector[pos] = caramelo;
        //insertamos en la primera posicion que se encuentre vacia del vector, la posicion del caramelo
        if (fila != 0) {//Adyacente de arriba
            ver_candy(mat, n, m, colum, fila - 1, vector,elemento);
        }
        if (fila != n - 1) {//Adyacente de abajo
            ver_candy(mat, n, m, colum, fila + 1, vector,elemento);
        }
        if (colum != 0) {//Adyacente de la izquierda
            ver_candy(mat, n, m, colum - 1, fila, vector,elemento);
        }
        if (colum != m - 1) {//Adyacente de la derecha
            ver_candy(mat, n, m, colum + 1, fila, vector,elemento);
        }

    }
}
```

Imagen 1.4

Otra función cuyo funcionamiento debe ser comentado es la función del host “eliminar_elementos”, la cual realizará todos los preparativos para realizar llamadas al kernel, como ya se muestra en la IMAGEN 1.2.

Sin embargo, esta función hace distinción de cuantos elementos se van a borrar, en caso de que se borre un solo elemento, si este es un “caramelo” normal (del 1 al 6), se perderá una vida y se modificará la matriz con la función “eliminar1” que se encargará de sustituir el elemento seleccionado en la matriz por un -1 (elemento borrado).

Sin embargo, si se trataba de un “caramelo especial” (B, T, o Rx) no se restará una vida y se llamará a su respectiva función para que modifique la matriz según el efecto del caramelo. Estas funciones son “explotarBomba”, “explotarTNT” y “explotarRx” respectivamente.

```
__global__ void explotarBomba(int* mat, int n, int m, int fila, int columna,int tipo, int* vector) {
    int idx = threadIdx.x;
    int idy = threadIdx.y;

    if (tipo == 0) {//Eliminar la columna entera
        if (idx == columna) {
            mat[idy * m + idx] = -1;
        }
    }
    else {//Eliminar la fila entera
        if (idy == fila) {
            mat[idy * m + idx] = -1;
        }
    }
    //el elemento deja de ser analizado
    vector[0] = -1;
}
```

Imagen 1.5

Por otra parte, si se eliminan de 2 a 4 caramelos estos se eliminan simplemente sin ningún efecto especial con la función “eliminar_iguales_juntos”, en la cual cada hilo comprobará si su identificador está en el vector de posiciones que se tienen que eliminar, y en caso de que sí estén, cada hilo transformará esa posición de la matriz en un -1 (elemento eliminado).

```
__global__ void eliminar_iguales_juntos(int* mat, int n, int m,int* vector) {
    // Calcular las coordenadas x e y del hilo
    int idx = threadIdx.x;
    int idy = threadIdx.y;

    for (int i = 0; i < n * m; i++) {
        if (vector[i] == idy * m + idx) {
            mat[idy * m + idx] = -1;
            vector[i] = -1;// El número está presente en el vector
            break;
        }
    }
}
```

Imagen 1.6

Otros casos son, el que se eliminen 5 elementos, el que se eliminen 6 o que se eliminen 7 o más. Por cada caso existe una función kernel que se encarga de eliminar todas las posiciones, pero a la diferencia de las anteriores, el elemento donde empezó la adyacencia (caramelo escogido por el usuario en modo de juego manual, o aleatorio en modo de juego automático) no se borrará, sino que será sustituido por una bomba, una TNT o un bloque rompecabezas respectivamente.

Estas funciones son eliminar5, eliminar6 y eliminar7oMas.

```
__global__ void eliminar5(int* mat, int n, int m, int* vector,int fila,int columna) {
    // Calcular las coordenadas x e y del hilo
    int idx = threadIdx.x;
    int idy = threadIdx.y;

    if (vector[0] == idy * m + idx) {
        mat[vector[0]] = 7;//Se pone una bomba en la posicion
        vector[0] = -1;
    }

    for (int i = 1; i < n * m; i++) {
        if (vector[i] == idy * m + idx) {
            mat[idy * m + idx] = -1;
            vector[i] = -1;// El número está presente en el vector
            break;
        }
    }
}
```

Imagen 1.7

Por otra parte, tenemos la función que se encargará de simular la gravedad entre los bloques, es decir, si un caramelo tiene algún boque por debajo que haya sido eliminado, este deberá “caer” de forma que los elementos eliminados permanezcan arriba de su columna, mientras el resto de elementos estén abajo conservando el orden que estaba anteriormente sin contar los elementos que han sido borrados.

Esta función funciona de manera que cada hilo debe comprobar cuantos elementos eliminados hay por debajo de este, una vez que haya contado cuantos tiene por debajo, intercambiará su posición por el de n (número de elementos borrados encontrados en su misma columna por debajo de este) posiciones por debajo. De esta manera se reordenar los caramelos según el nuevo orden necesario.

```
__global__  void caer_caramelos(int* matriz, int n, int m) {
    // Calcular las coordenadas x e y del hilo
    int idx = threadIdx.x;
    int idy = threadIdx.y;


    // Contar los elementos -1 debajo del hilo
    int num_minus_1 = 0;
    for (int i = idy; i < n; ++i) {
        if (matriz[i * m + idx] == -1) {
            num_minus_1++;
        }
    }
    __syncthreads();
    //Intercambiar la posicion de n veces abajo
    if (num_minus_1 > 0 && matriz[idy * m + idx] != -1) {
        int aux = matriz[idy * m + idx];
        matriz[idy * m + idx] = -1;
        matriz[(idy + num_minus_1) * m + idx] = aux;
    }
}
```

Imagen 1.8

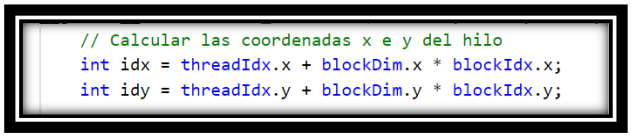
Por último, está la función que sustituirá los elementos borrados por unos nuevos aleatorios. Esta función es “rellenar_huecos”, esta función es muy parecida a la función “matriz_aleatoria” mostrada en la IMAGEN 1.3, con la diferencia de que solo los hilos cuya posición en la matriz sean elementos eliminados (el número -1).

Tras la ejecución de todas estas funciones se cumpliría un ciclo del juego, el cual no acabará hasta que el usuario pierda todas las vidas.

Implementación optimizada

Tablero finito con múltiples bloques

En este apartado se debe adaptar el código del apartado anterior para que pueda correr con múltiples bloques. Para esto no se deben modificar en gran medida las funciones ya existentes. En esencia lo único que se cambia es el índice de los hilos, ya que ahora se debe tener en cuenta al bloque al que pertenece. A continuación, se mostrará una imagen que se podrá comparar a la IMAGEN 1.3 para poder visualizar las diferencias de cómo se inicializan los hilos.

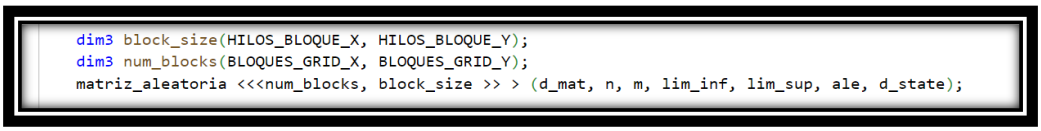


```
// Calcular las coordenadas x e y del hilo
int idx = threadIdx.x + blockDim.x * blockIdx.x;
int idy = threadIdx.y + blockDim.y * blockIdx.y;
```

Imagen 2.1

Por otra parte, a la hora de inicializar la llamada al kernel se ha implementado una función para que calcule el número de hilos y el número de bloques, para así obtener la mayor optimización posible.

De esta manera se obtienen los valores óptimos para la ejecución del programa. Tras esto, se lanzan las respectivas llamadas al kernel, que gracias a la distinción que se ha realizado en la IMAGEN 1.10, podemos observar cómo se lanzan los kernel.



```
dim3 block_size(HILOS_BLOQUE_X, HILOS_BLOQUE_Y);
dim3 num_blocks(BLOQUES_GRID_X, BLOQUES_GRID_Y);
matriz_aleatoria <<<num_blocks, block_size >> > (d_mat, n, m, lim_inf, lim_sup, ale, d_state);
```

Imagen 2.2

En dicha función (mejoresCaracterísticas), obtenemos mediante funciones CUDA las características, valga la redundancia, de la tarjeta gráfica del ordenador del usuario ejecutor del programa. En base a esas características y al tamaño de la matriz que crea, establecemos el número de hilos por bloque y número de bloques con que se lanzan las funciones kernel.
De esta forma, el ThreadsPerBlock.x* ThreadsPerBlock.y*BlocksPerGrid.x* BlocksPerGrid.y, deben ajustarse lo máximo posible a N*M (el tamaño de la matriz).

En algunas situaciones puede ser más eficiente lanzar un solo bloque con muchos hilos. Esto puede ocurrir, por ejemplo, cuando se está trabajando con matrices muy pequeñas (como es el caso de esta práctica) y el tiempo necesario para lanzar múltiples bloques y sincronizarlos supera al tiempo de cálculo real. En estos casos, lanzar un solo bloque con muchos hilos puede ser más rápido.

Tablero finito con múltiples bloques y memoria compartida entre estos

Este último apartado es similar al anterior, con la diferencia de que en algunas funciones kernel, abordamos su desempeño utilizando memoria compartida entre los distintos hilos del bloque.
Por tanto, en las funciones que hemos estimado necesario (todas en las que se modifica el vector y sustancialmente la matriz y que mejora la eficiencia),

Un ejemplo de esto puede ser cualquier función de eliminar, como eliminar6. Donde copiamos el vector a memoria compartida para que al hacer la búsqueda de todas sus posiciones en busca de una coincidencia entre un elemento del vector y el id de nuestro hilo, los hilos accedan a memoria compartida y estos accesos sean mucho más rápidos.

```
__global__ void eliminar6(int* mat, int n, int m, int* vector, int fila, int columna) {
    // Definir la memoria compartida
    __shared__ int s_vector[SH_DIM_X * SH_DIM_Y];

    // Calcular las coordenadas x e y del hilo
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    int tid = threadIdx.x + threadIdx.y * blockDim.x;

    bool centinela = false;
    int pos = fila * m + columna;

    // Copiar el vector a la memoria compartida
    if (tid < n * m) {
        s_vector[tid] = vector[tid];
    }
    __syncthreads();

    if (s_vector[0] == idy * m + idx) {
        mat[fila * m + columna] = 8;//se pone una TNT en la posicion
        s_vector[0] = -1;
    }

    for (int i = 1; i < n * m; i++) {
        if (s_vector[i] == idy * m + idx) {
            centinela = true; // El número está presente en el vector
            pos = i;
        }
    }

    // Verificar si el hilo se encuentra dentro de los límites de la matriz y coincide con una posición que hay que eliminar
    if (centinela) {
        mat[idy * m + idx] = -1;
        s_vector[pos] = -1;
    }

    // Copiar el vector de vuelta a la memoria global
    if (tid < n * m) {
        vector[tid] = s_vector[tid];
    }
}
```

Imagen 2.3

La matriz no la hemos realizado en esta función de forma compartida también porque en el peor caso, solo se accede una vez a ella para eliminar un elemento, por lo que el coste de copiarla a memoria

compartida y después copiarla de vuelta a memoria normal, consideramos que es superior al de acceder una sola vez a un elemento de la matriz.

Mejoras realizadas

Respecto a la interfaz del usuario, hemos tuneado bastante la función imprimir para que, como hemos dicho antes, mejore la jugabilidad.
Los cambios hechos aquí son:

- Indicación de índices de las filas y columnas para facilitar la identificación de la posición del elemento a seleccionar y evitar tener que estar contando casillas.
- Color a los distintos números según el enunciado, con la excepción del naranja que ha sido modificado por lila ya que había cierta similitud entre naranja, amarillo y marrón.
- Color a los Rx, cada uno relacionado con su correspondiente número x.
- Los caramelos especiales parpadean, para destacar más su presencia.