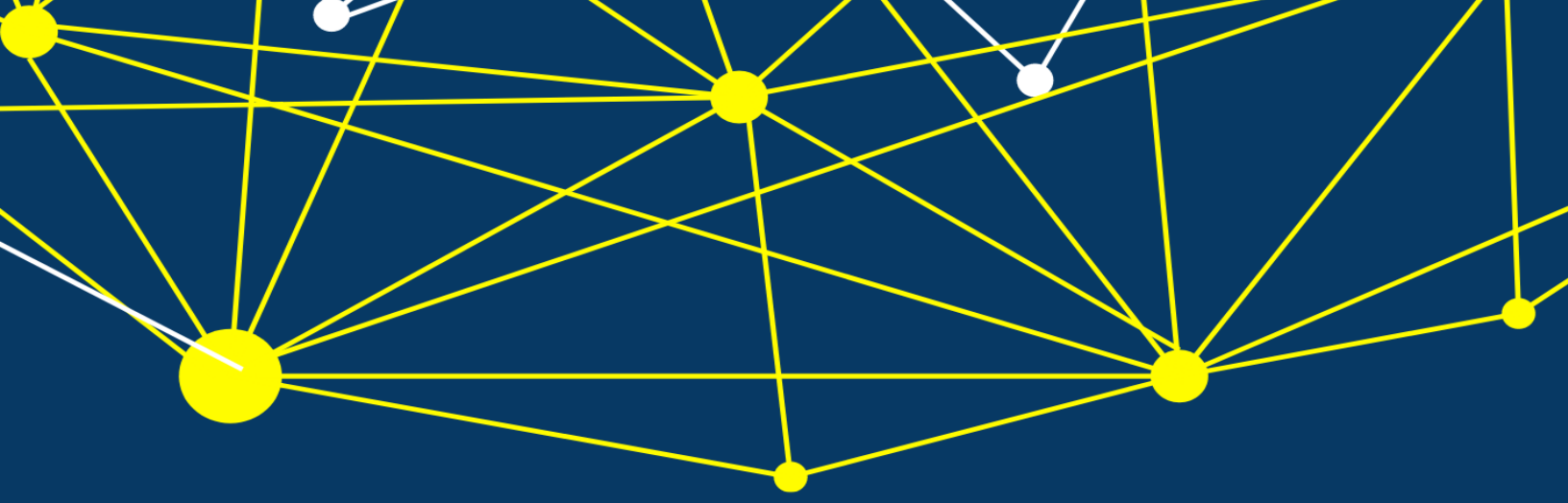

Paradigmas Avanzados de Programación, 3ºGII

Daniel de Heras Zorita

Adrián Borges Cano

PECL1: CUDA

ÍNDICE

Tabla de contenido

Introducción

Este documento ha sido creado para facilitar la explicación del programa desarrollado en CUDA, un lenguaje de programación, muy relacionado con C, creado por nVidia, factor que nos obliga a tener una tarjeta gráfica de la marca nVidia para poder programar en dicho lenguaje. La práctica trata sobre la realización de un juego llamado 'Cundy Crosh Soga', una adaptación al famoso juego de Smartphone 'Candy Crush Saga'. Este trabajo llevado a cabo por Daniel de Heras Zorita y Adrián Borges Cano, corresponde a la PECL1 de Paradigmas Avanzados de Programación, de 3ºGII de la UAH, cuya entrega corresponde al 31 de marzo de 2023.

En este documento explicaremos las distintas versiones que hemos debido generar para cumplir con lo requerido, al igual que una descripción clara sobre el funcionamiento de cada una de las funciones realizadas, en caso de no saber cual es su propósio, y querer conocerlo. Sin más dilación, comencemos a adentrarnos en el proyecto.

Versiones del proyecto

Antes de comenzar a hablar sobre lo codificado, debemos primero entender qué debíamos entregar para poder tener un hilo del que tirar, sobre todo al comienzo de la práctica.

En este trabajo, se nos requería entregar 3 proyectos distintos, donde su funcionalidad fuese la misma en todos los casos, pero la forma de implementarlo fuese diferente en cada una de ellas. Para poder hablar sobre las distintas implementaciones, primero debemos hacer un pequeño análisis al funcionamiento de CUDA.

Como se ha comentado en la introducción, CUDA es un lenguaje de programación que obliga al usuario el tener en su poder una tarjeta gráfica, concretamente de la marca nVidia. Este requerimiento es esencial, porque es el corazón del funcionamiento de dicho lenguaje. CUDA es una herramienta que sirve para agilizar las operaciones que normalmente se hacen en el procesador del ordenador, dado un determinado programa, enviando dichas operaciones a la tarjeta gráfica del sistema, puesto que esta contiene mayor ancho de banda, cosa que le permite realizar con mayor velocidad cálculos sencillos que en la CPU llevaría más tiempo. Por tanto, la manera que tiene CUDA para indicar cuantos recursos de nuestra GPU queremos utilizar para una función es mediante la indicación de la **dimensión del grid** a utilizar (números de bloques al que llamaremos) y la **dimensión del bloque** (número de hilos que contendrá cada uno de los bloques llamados). Esto se hace de la siguiente forma:

```
funcionEjemplo <<<numBloques, numHilos>>> (parametrosFuncionEjemplo);
```

Figura 1: Declaración de una función en CUDA

Tanto `numBloques` como `numHilos` pueden ser objetos de tipo `dim3`, pero no es necesario entrar en tanto detalle por el momento. Aunque la comunicación entre los hilos de un mismo bloque es trivial, puesto que no se debe hacer nada para conseguirla, la comunicación entre bloques, por defecto, no existe. Esto significa que para poder comunicar distintos bloques en CUDA, se debe usar el concepto de **memoria compartida**, aunque profundizaremos en ello más adelante.

Una vez realizada esta breve explicación, y se conocen los conceptos de bloque e hilo, podemos hablar sobre lo pedido. En esta entrega, se pedía aportar 3 proyectos de CUDA distintos.

Implementación básica

Lo único que se pedía era implementar todo lo pedido en los requisitos (distintas características que debía contener el juego) de la forma más sencilla posible: usando únicamente 1 bloque en cada ejecución de los métodos que son llamados al *device* (lo que es equivalente a la tarjeta gráfica). En este caso, el número de hilos debe ser menor que los hilos que soporte el bloque, número que será relativo, ya que depende de la GPU del usuario. Por lo tanto, aunque el programa funcione igualmente, no lo hace de la forma más rápida posible ya que, si usásemos más de un bloque en cada función, podríamos permitirnos asignar operaciones más sencillas a cada hilo ya que, al tener más, estos se dividirían el trabajo, acelerando el tiempo de ejecución del programa.

De esta forma, desde el punto de vista del programador, este apartado es el más sencillo, puesto que no debe preocuparse en pensar lo que debe hacer cada uno de los hilos, porque es dicho hilo el encargado de hacer las tareas del método. Él es el único encargado de esa operación, sin necesidad de compartirlo con ningún otro hilo. Sin embargo, desde el punto de vista del rendimiento, los recursos proporcionados por la tarjeta gráfica no están siendo ni mucho menos aprovechados, ya que utilizar un único bloque para la gran capacidad computacional que tiene cualquier gráfica de nVidia significa no aprovechar todo el potencial disponible. Recuerda a la metáfora de tener un Ferrari e ir por carreteras con límite de 30Km/h.

Ejercicio 7.- Implementación optimizada con múltiples bloques

Debemos confesar que como comenzamos a desarrollar el juego antes de centrarnos plenamente en lo que debíamos entregar, la forma inicial en la que enfocamos la práctica fue la que se pide en este apartado. Desde un primer momento, planeamos todas las ejecuciones de los métodos de forma que fuese usado por múltiples bloques, debido a que muchos de los métodos programados, como el de sustituir los bloques vacíos del tablero por nuevos números aleatorios, están ideados de forma que un hilo acceda a una única posición del hilo. Por lo tanto, necesariamente necesitábamos más bloques para poder asegurar que se accediesen a todas las posiciones del tablero. Como se ha comentado anteriormente, dicho número de bloques va a depender de los recursos Hardware de los que goce el usuario.

En caso de haber seguido un orden lineal con respecto a lo pedido en el enunciado, deberíamos haber modificado los métodos para mejorarlos y que pudiesen ser usados por más de un bloque. No obstante, como hemos comentado, nuestro proceso de adaptación fue mucho más sencillo, ya que hicimos el proceso inverso. Creamos un nuevo proyecto teniendo como base el del ejercicio 7, adaptando las funciones para que fuesen usadas por un único bloque.

Ejercicio 8.- Implementación optimizada con múltiples bloques y memoria compartida

En este caso, ocurría lo mismo que lo comentado en el apartado anterior. Gracias al haber desarrollado con éxito el uso de múltiples bloques, este ejercicio se simplificaba

notablemente, puesto que solo debíamos encargarnos de analizar qué métodos debían usar memoria compartida, para que hiciesen uso de ella todos aquellos métodos que pudiesen gozar de ella.

Debemos explicar que el único método que hace uso de la memoria compartida es el de `eliminarBloques`. A pesar de haberlo intentado, en el resto de ellos hemos considerado que el uso de esta memoria era inviable, puesto que algunos de ellos no lo permitían, mientras que otros de ellos, en caso de permitirlo, si se hubiese implementado traería consigo más inconvenientes que ventajas. En adición, para que el lector pueda comprender por qué el resto de los métodos no pueden utilizar este tipo de memoria, le invitamos a que revise el siguiente apartado, en caso de que quiera reflexionar si realmente se puede usar memoria compartida en alguna función más, sin contar la ya mencionada.

Métodos implementados

A continuación, haremos un repaso por todos los métodos codificados en la práctica. Cabe recalcar que como lo único que cambia entre proyectos es el número de bloques usados, las funciones son equivalentes para todos. Si bien es cierto que quizá haya diferencias entre los mismos métodos de proyectos distintos para satisfacer lo pedido, estos cambios son despreciables a la hora de entender el funcionamiento de estos. Igualmente, en caso de haberlos, serán indicados.

Métodos en el Device

En este apartado hablaremos sobre los métodos que han sido creados para que se ejecuten la tarjeta gráfica del usuario.

rellenarTablero

Dado un tablero, buscar qué posiciones tienen valor 0, y en caso de tenerlo, sobrescribirlo por un número aleatorio entre los caramelos disponibles.

- Implementación Básica: cada hilo del único bloque se encarga de rellenar los valores de una columna.
- Implementación optimizada: cada hilo se encarga de acceder únicamente a una posición, ahora desde distintos bloques.

eliminarBloques

Dada una posición seleccionada dentro de la matriz, eliminar todos lo bloques adyacentes que tengan el mismo valor que el seleccionado. Para ello, busca tanto en el entorno de su fila como de su columna los bloques que sean igual a dicha posición. Para eliminarlos, los sobrescribe a 0.

- Implementación Básica: el único bloque disponible es el que maneja toda la matriz.
- Implementación optimizada: cada bloque accede a una fila o una columna de la matriz, y los hilos comprueban las adyacencias dentro de cada una.
- Implementación con memoria compartida: como sabemos que cada bloque va a acceder únicamente a filas o columnas, creamos una fila y columna compartidas en las que meteremos los elementos a los que accedería en su ejecución normal. Así, todos los hilos realizaran las múltiples comprobaciones accediendo a memoria compartida, aumentando el rendimiento y reduciendo el tiempo de acceso.

activarBomba

Dada la posición de la matriz que se quiera borrar, selecciona aleatoriamente la fila o columna correspondiente a dicha posición, y la recorre eliminando uno a uno cada elemento. Para eliminarlos, los sobrescribe a 0.

- Implementación básica: cada hilo elimina únicamente una posición de la fila/columna.
- Implementación optimizada: usando varios bloques, cada hilo elimina únicamente una posición de la fila/columna.

activarRompecabezas

Dado el color del bloque que se quiera eliminar, representado como un número, buscar todas las apariciones de dicho color en la matriz, y en caso de encontrar una aparición, borrarla. Para poder eliminarlo, lo sobrescribe a 0.

- Implementación básica: cada hilo accede a una posición, comprueba y elimina.
- Implementación optimizada: usando varios bloques, cada hilo accede únicamente a una posición, comprueba si es igual que el color asignado, y en caso de serlo, lo borra.

activarTNT

Dada la posición de la matriz en la que se quiere activar el TNT, eliminar todos los bloques que se encuentren en un rango de 4 elementos. Para ello, si la posición seleccionada es (X,Y), todas las coordenadas que se encuentren entre el rango de (X-4,Y-4) y (X+4, Y+4) serán eliminadas. Para poder eliminarlas, se sobrescriben a 0.

- Implementación Básica: cada hilo comprueba si puede eliminar el elemento de su posición.
- Implementación optimizada: usando varios bloques, cada hilo accede a una posición de la matriz. Si dicha posición se encuentra en el rango indicado, entonces borrará el bloque.

dejarCaerBloques

Dado el tablero del juego, recorrer por columnas en busca de ceros. Dichas columnas son recorridas de arriba abajo. En caso de que encuentre alguno, lo intercambia iterativamente con el elemento que tiene encima hasta llegar a la primera fila de la matriz. Por tanto, lejos de bajar los bloques de colores al fondo de la matriz, se están subiendo los bloques vacíos a lo alto de la misma. A continuación, en la Figura 2 se puede observar un ejemplo de la secuencia en el tiempo que sigue este algoritmo para dejar caer todos los bloques de colores.

5	5	~~5~~ 0	0	0	0	0	~~0~~ 0	0
2	~~2~~ 0	~~0~~ 5	5	5	5	~~5~~ 0	~~0~~ 0	0
0	~~0~~ 2	2	2	2	~~2~~ 0	~~0~~ 5	5	5
3	3	3	3	~~3~~ 0	~~0~~ 2	2	2	2
0	0	0	0	~~0~~ 3	3	3	3	3
1	1	1	1	1	1	1	1	1
Paso1	Paso2	Paso3	Paso4	Paso5	Paso6	Paso7	Paso8	Paso9

Figura 2: Secuencia en el tiempo de cómo se ejecuta 'dejarCaerBloques'

En esta figura, se ha marcado en color verde el cero por el que se comienza a iterar. En rojo están marcadas todas las sustituciones realizadas. Finalmente, en azul se muestra el resultado final del algoritmo.

- Implementación básica: se lanza un hilo por columna, en un único bloque.
- Implementación optimizada: cada bloque se encarga de recolocar los elementos de su columna asignada.

ponerPowerup

Este es el método encargado de colocar un *Powerup* (potenciador) en el lugar que corresponda. Las opciones posibles son: bomba (B), TNT (T) y rompecabezas (Rx, donde *x* es el color del bloque). Para ello, buscamos en la matriz el número de ceros que haya en la matriz. Cada vez que se encuentre un cero, se incrementa atómicamente (`atomicAdd`) en uno la variable `posicionesCero`. En caso de que dicha variable tenga valor 5, entonces se colocará una bomba. Como en CUDA no podemos mezclar números y letras en una matriz de números, lo hemos guardado con valor 10. En caso de encontrar 6 ceros, entonces se guardará un 20, haciendo referencia al TNT, mientras que si hay 7 o más, entonces se guardará un rompecabezas, con valor 50 sumado al color del caramelo, para poder imprimir posteriormente el color del mismo.

- Implementación Básica: cada hilo accede, comprueba e incrementa en uno en caso de que sea posible.
- Implementación Optimizada: usando varios bloques, cada hilo accede únicamente una posición de la matriz, e incrementa en uno.

Métodos en el Host

A continuación, hablaremos sobre los métodos que se ejecutan en la propia CPU del usuario. Esta parte es mucho más simple, puesto que no se necesita hacer continuas llamadas al kernel (núcleo de la GPU, encargado de realizar las operaciones) de la GPU.

validate_input
Como lo que se recibe de lo que ha introducido el usuario por teclado es un carácter, primero comprobamos que equivalga a un número, y en caso afirmativo, devolverlo para poder manejar con él.

posicionesEliminadas
Comprobar en la matriz del tablero cuantas posiciones se han eliminado, y devolverlo como parámetro de salida.

print_matrix
Recorrer la matriz secuencialmente para imprimirla con el formato querido. Este formato corresponde al siguiente:

Valor en la matriz	Impresión
0	*Vacío*
10	B
20	T
50-56	R*x*
1-6	1-6

Figura 3: Formato de impresión del tablero

Debemos aclarar un par de cosas sobre la Figura 3. Por un lado, en caso de encontrarnos con un cero, la matriz devuelve una posición vacía, representada con espacios, de forma que cuando ejecutemos métodos como el de `dejarCaerBloques`, este se verá mucho más realista. Por otra parte, en caso de encontrarnos con un valor entre 50 y 56, entonces sabremos que es un rompecabezas. En adición, para saber de qué color es dicho rompecabezas, hacemos el módulo en base 10 de dicho número, obteniendo el color, que es lo que se corresponde con *x*.

Ejecución del programa
En este apartado, mostraremos cómo está estructurado el funcionamiento del programa, explicando los métodos por los que entra y sale el mismo para poder llevar a cabo el funcionamiento del juego. Además de esto, nos apoyaremos en las funciones explicadas del apartado anterior para permitir al lector entender lo mejor posible este flujo de ejecución.

Antes de comenzar a jugar, debemos abrir el fichero ejecutable que contiene la práctica. A la hora de ejecutarlo, existen 2 formas de hacerlo:

1.- **Indicando distintos parámetros por consola**: estos parámetros son 4.

- Modo de ejecución. En caso de seleccionar `-m`, entonces el usuario podrá usar el Coundy Crosh, de lo contrario, si introduce `-a`, el programa se ejecutará de forma automática. El usuario únicamente tendrá que clicar la tecla `ENTER` cada vez que quiera avanzar hacia la siguiente acción.

- Nivel de dificultad: Para ello, existen dos modos: fácil, introduciendo `1`, que tiene bloques del 1 al 4; y difícil, introduciendo `2`, que tiene bloques del 1 al 6.
- Tamaño del tablero: En este caso, tenemos 2 parámetros: `numColumnas`, `numFilas`. Como se puede imaginar, cada uno corresponde al número de columnas y filas que contendrá la matriz, respectivamente.

En la siguiente figura se muestra un ejemplo de cómo se puede ejecutar el programa.

```
CandyCroshej7.exe -m 1 10 10
```

Figura 4: Ejemplo de ejecución del programa por consola con parámetros

2.- **Ejecutando directamente el programa:** En ese caso, los parámetros mencionados en el punto 1 deberán ser pedidos por consola.

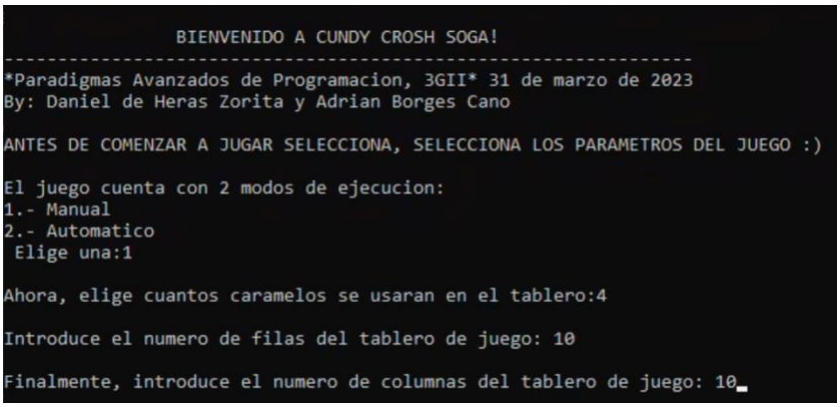

Figura 5: Ejemplo de ejecución del programa por consola sin parámetros

Una vez inicializado el programa, se nos presentará la matriz correspondiente al tablero del juego. ¡Ya se ha comenzado a jugar! Para que esto sea llevado a cabo, se genera una matriz llena de ceros con las dimensiones que haya indicado el usuario. Posteriormente, se llama a `rellenarTablero` para completarlo con bloques de colores. Gracias a este método y esta implementación, llamándole podemos llenar los huecos con ceros siempre que lo necesitemos.

De ahora en adelante, explicaremos cómo funciona la ejecución del programa en caso de que el usuario hubiese seleccionado la opción de juego manual, ya que la automática es igual, solo que más simplificada.

A partir de ahora, el programa le pedirá constantemente al usuario que introduzca las coordenadas del bloque que quiera eliminar.

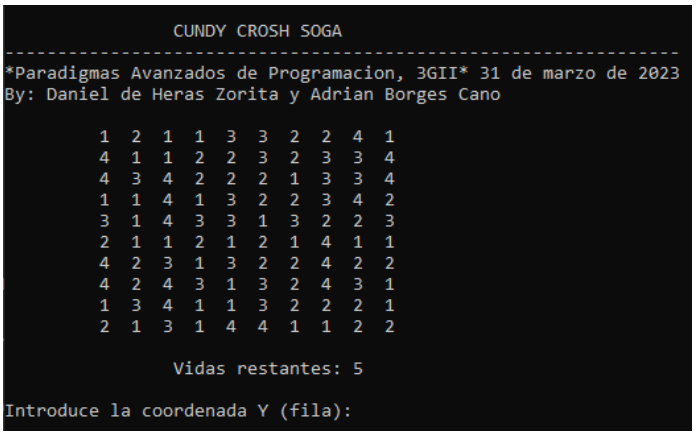

Figura 6: Primera impresión del juego

Dependiendo del bloque que introduzca el usuario, se pueden dar distintos casos:

1.- **Los bloques eliminados no provocan ningún potenciador**: Los bloques serán eliminados, el número de vidas se mantendrá igual, y se generarán nuevos bloques para ocupar los espacios vacíos.

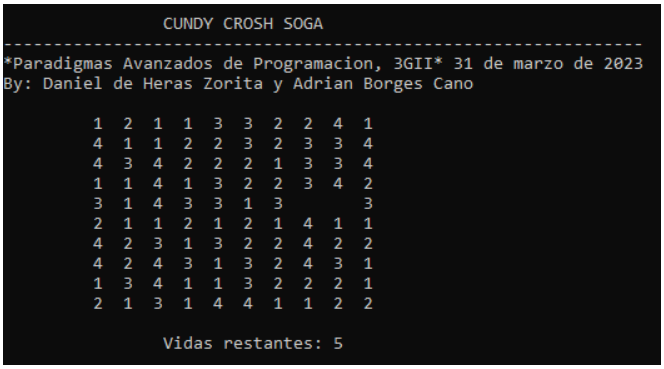

Figura 7: Bloques vacíos

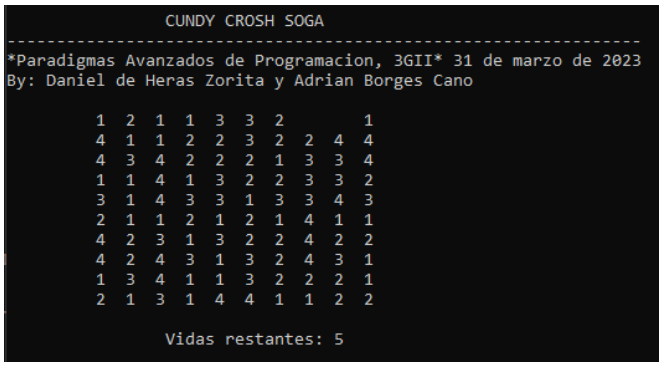

Figura 8: Bloques bajados

2.- **Los bloques eliminados provocan una bomba**: si el jugador la selecciona, se eliminará aleatoriamente su fila o columna correspodiente.

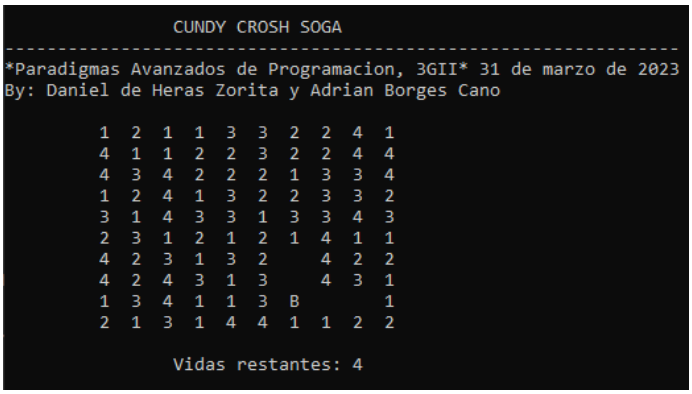

Figura 9: Bloques vacíos con Bomba

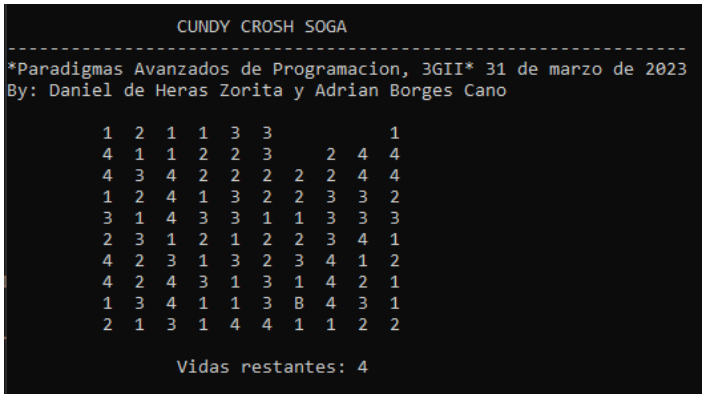

Figura 10: Boques bajados con bomba

La ejecución de Figura 9 y Figura 10 es equivalente ante cualquier potenciador. Por ello, a partir de ahora se omitirá.

Figura 11: Bloques eliminados con bomba

Figura 12: Bloques bajados al activar bomba

3.- **Los bloques eliminados provocan un bloque de TNT**: si el jugador lo selecciona, se eliminarán todos los bloques en un radio de 4 elementos.

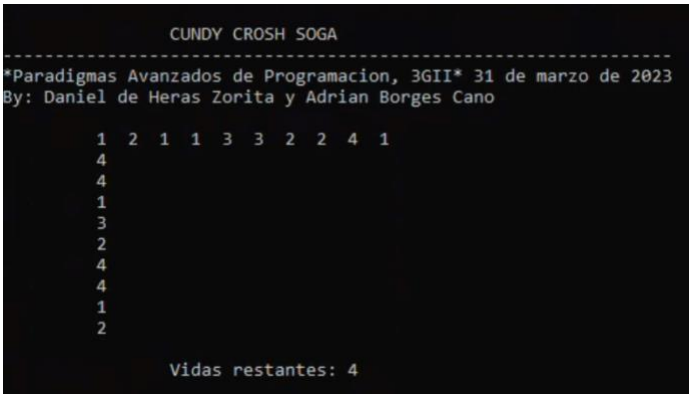

Figura 13: Bloques eliminados con TNT

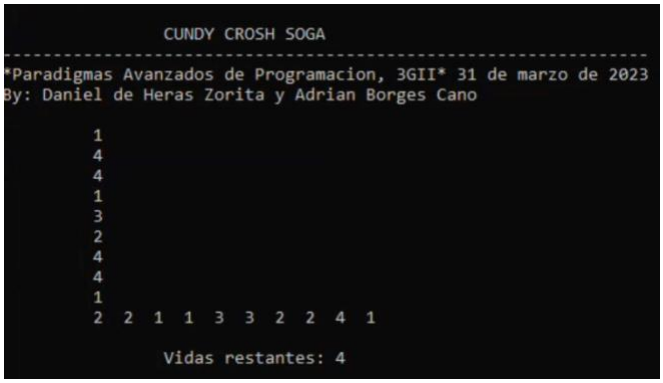

Figura 14: Bloques bajados al activar TNT

4.- **Los bloques eliminados provocan un rompecabezas**: si el jugador la selecciona, eliminará todos los bloques que tengan el mismo color que dicho rompecabezas.

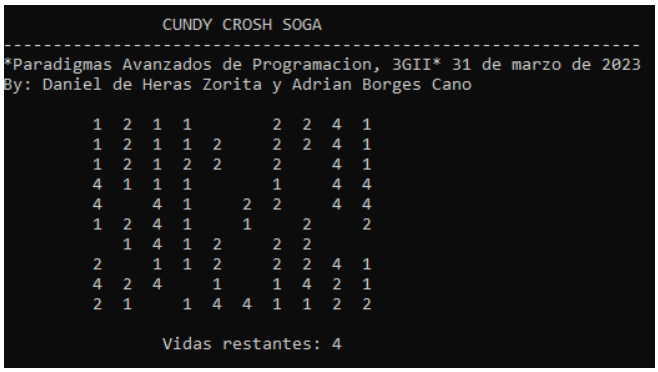

Figura 15: Bloques eliminados con rompecabezas

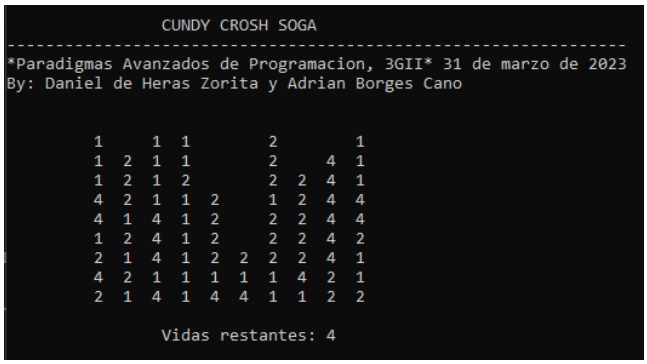

Figura 16: Bloques bajados después del rompecabezas

5.- **No se pueden eliminar bloques en la posición indicada**: se mostrará por pantalla que no se ha podido hacer ninguna combinación posible, y se restará una vida.

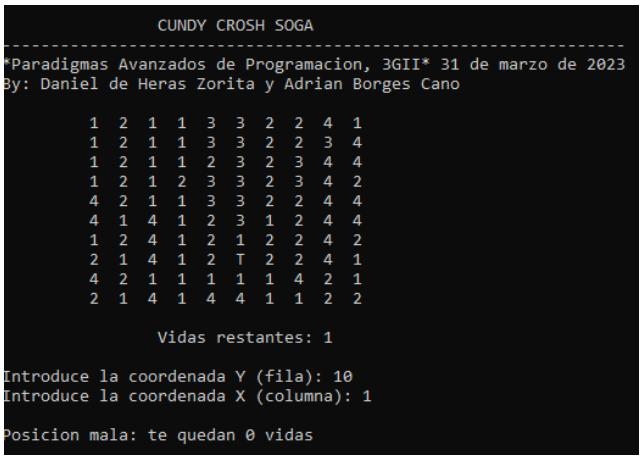

Figura 17: No hay combinación

Combinando todos los puntos anteriormente comentados, el usuario podría permitirse jugar durante horas y, si es habilidoso, mantenerse con suficientes vidas como para seguir jugando.

Finalmente, si el usuario agota las vidas, el juego terminará.

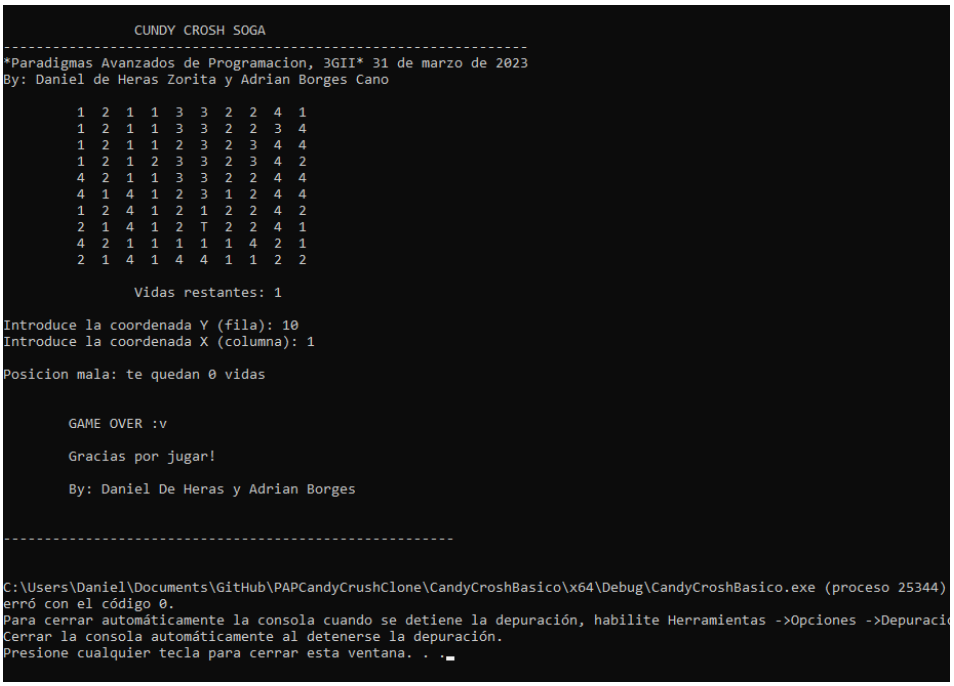

Figura 18: Pantalla final del juego

Una vez se ha terminado la ejecución del programa (al haber agotado las vidas), se libera la memoria reservada en la GPU que ha sido usada para enviar de un procesador a otro la matriz que corresponde al tablero.

Mejoras realizadas

Aunque no fuese pedido, hemos añadido un par de mejoras en la **interfaz** del proyecto, para poder presentar el mismo de una manera más llamativa, pero sobre todo, que sea lo más cómoda posible para el usuario.

Para ello, hemos implementado 2 ideas principales. Estas ideas son: **colorear** los caramelos con su color correspondiente. Siguiendo el formato pedido en el enunciado, escribimos los bloques por pantalla con el color correspondiente a su número. **Puntuación** obtenida en el juego. Para que el jugador pueda saber cómo de buena ha sido su partida, tenemos un marcador que cuenta cuantos puntos ha obtenido durante la partida. Cada bloque eliminado equivale a 10 puntos. Al terminar la partida, muestra la puntuación final del mismo para que el usuario tenga constancia de su resultado.

A continuación, mostraremos una captura con ambas implementaciones realizadas.

Figura 19: Coloración de los bloques

Como se ha comentado, al terminar la partida, el usuario puede saber qué puntuación ha obtenido.

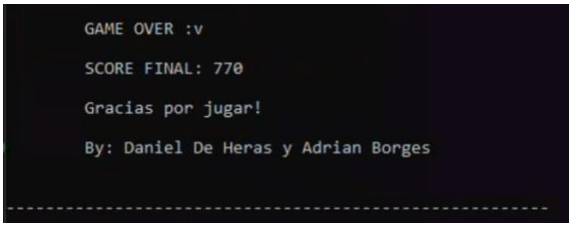

Figura 20: Impresión de la puntuación final

Conclusiones

Finalmente, después del duro trabajo realizado durante varias semanas, podemos dar por finalizada dicha práctica. Debemos decir que, a pesar de ya haber hecho el examen de teoría, creemos que esta práctica ha sido mucho más útil para permitir afianzar los conceptos en lo que CUDA se refiere. Gracias a esto, hemos conseguido obtener una mejor visión espacial sobre el funcionamiento interno de los componentes de un ordenador a la hora de ejecutar un programa, al igual que ser capaces de aprovechar sus recursos de gran forma en la medida de lo posible. Esperemos que haya sido de agrado para el lector esta memoria, y que haya disfrutado del proyecto ejecutado, tanto dentro como fuera del código de CUDA.