Ampliación de programación avanzada. 2048 CUDA

Juan Casado Ballesteros.

Casandra Moreno.

Índice

1. Introducción 3
2. Planteamiento del juego 3
3. Límites y Dificultades 8
4. Partes realizadas 8
5. Inteligencia Artificial 8
6. Funciones de la GPU 9
   - 6.1. Void flipH (float *tablero, int size, int nc) 9
   - 6.2. Void flipV(float *tablero, int size, int nc) 9
   - 6.3. void createDeleterV(float *tablero, float *decisions, float *out, int size, int nc) 9
   - 6.4. void moveH(float *tablero, float *decisions, int size, int nc) 9
   - 6.5. void moveV(float *tablero, float *decisions, int size, int nc, int nf) 9
   - 6.6. Void takeDecisionsH(float *tablero, float *decisions,int size, int nc) 9
   - 6.7. Void sumLeft(float* tablero, float* result,int size, int nc, int nf) 9
   - 6.8. Void sumPoints(float *decisions, float *sum_result, int size, int max_steps) 9
   - 6.9. Void sumGaps(float *decisions, float *sum_result, int size, int max_steps. 10
   - 6.10. Void sumMovements(float *decisions, float *sum_result, int size, int max_steps) 10
   - 6.11. Void cpyMatrix(float *matriz, float *copia, int size) 10
   - 6.12. Void hasChanged(float *matriz, float *copia, int size) 10
7. Funciones de la CPU 10
   - 7.1. Void addRandom (T *tablero, int elements, int len) 10
   - 7.2. sumArray(T *arr, int len) 10
   - 7.3. T maxArray(T *arr, int len) 10
   - 7.4. std::string printTablero(T *tablero, int n_columnas, int n_filas) 10
8. Main 10

1. Introducción

El juego se ha implementado en memoria global, en memoria por bloques y en memoria compartida. La principal característica del proyecto es que cada función o acción se ha realizado en kernels independientes, facilitando así la comprensión. Se procura que cada acción se realice en su propia matriz de modo que se favorece el "debugging" de código pues en cualquier momento se puede traer una matriz a host (cada matriz de device tiene una matriz host reservada para hacer esto) y así poder mostrarla para ver su estado antes y despés de pasar por un kernel.

- Se lleva un recuento de los puntos conseguidos por cada vida.
- Cuando acaba la partida se muestra la suma de todos esos puntos, así como la suma de todos los conseguidos a lo largo del uso de jugo (se hace uso de un archivo de texto)
- En todo momento se permite cargar pulsando c o guardar pulsando g una matriz para jugar con ella.
- También se permite cambiar de modo automático a manual pulsando m o terminar la partida antes de tiempo pulsando e.
- El tamaño del texto en la consola se ajusta en función del tamaño de la matriz.
- Se detecta cuando un movimiento no produce cambios en la matriz y no se deja hacerlo al usuario.
- Si no quedan movimientos por hacer se detecta también.
- Se leen las características de la gráfica para decidir el tamaño de la tesela de modo que se gasten los menos hilos posibles.

2. Planteamiento del juego

Se expone la forma de resolver el problema a partir de la realización de un movimiento hacia abajo:

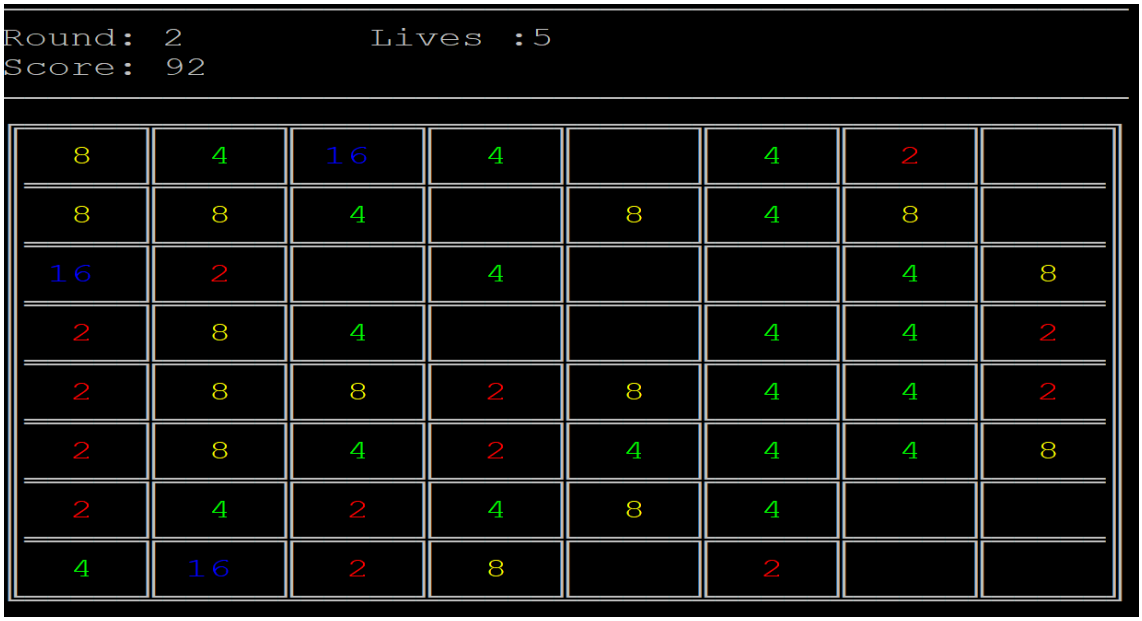
DATOS DE LA PARTIDA:

```
Multiprocesor count: 12
Max Threads per multiprocesor: 2048
Max Threads per block: 1024

Modo de ejecucion [ a | m ]: m
Cuantos elementos iniciales quiere [ 1 = 8 | 2 = 15 ]: 20
Introduzca el numero de filas del tablero: 8
Introduzca el numero de columnas del tablero: 8

Columnas : 8 | Filas: 8 -> Elementos: 64 | Max recursion: 6
Modo: manual | Elementos iniciales: 20
```

Se da la opción de configurar la partida cumpliendo los requisitos solicitado e incluso dando mayor libertad como poder poner cualquier tamaño de semilla

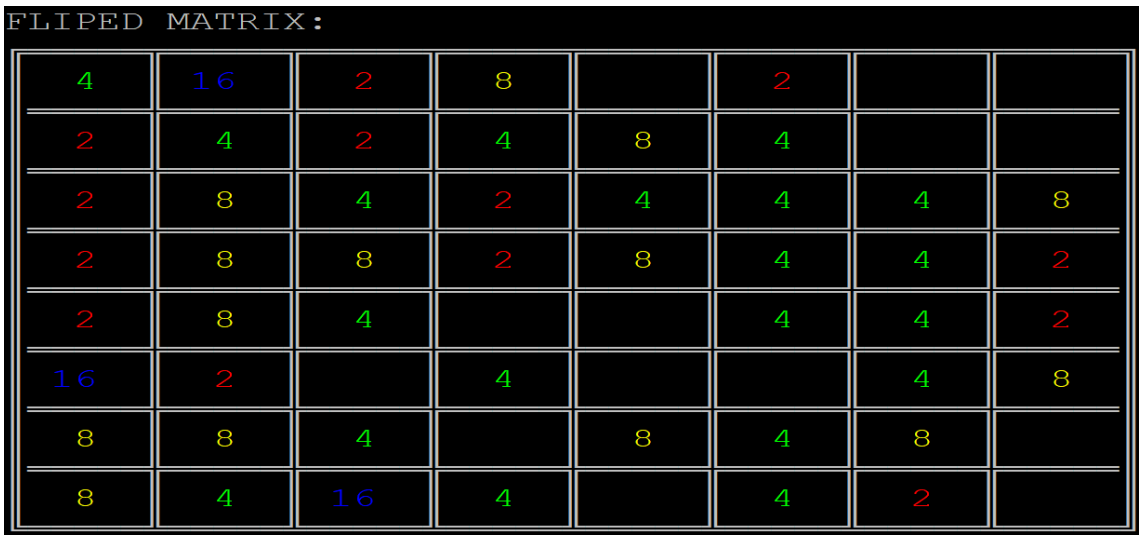
MATRIZ INICIAL

Para representar los datos se utiliza la consola con una ascii art simple que se adapta al tamaño de la matriz de juego de modo que las matrices pequeñas se vean ocupando toda la pantalla y que las más grandes entren en ella.



FLIP

Se da la vuelta a una matriz, solo tenemos dos tipos de kernel para hacer los vomientos, unos kernel cooperan para acabar haciendo un movimiento a la izquierda y otros hacia arriba de modo que para hacer un movimiento hacia abajo el primer paso es invertir la matriz por el eje correspondiente.



TOMA DE DECISIONES

Se crea la matriz de decisiones en la cual se indican las nuevas celdas (resultado de la unión de dos) que se generarán al hacer un movimiento.

Deja el valor que se obtendrá tras añadir dos elementos en la posición del elemento que se va a añadir

La toma de decisiones sirve también para saber los puntos que se obtiene al hacer el movimiento y contar los movimientos realizados. Se pondrá el valor a obtener en elementos que sumen con otro ocupan un lugar impar contando solo los ocupados por elementos iguales desde el primero que no es igual a ellos

Ej:

Tablero de entrada: 2222 400404

Matriz de decisiones: 0404 000800

Se pone la nueva ficha en las posiciones impares cada vez que se encuentra un número par de valores sumables

MATRIZ DECISIONES:

		4					
4					8		
	16		4			8	
4					8		4
						8	
		8		16			
16			8		8		

CREAR MATRIZ DE BORRADO

Se crea una matriz con un 1 en las posiciones del tablero que haya que poner a 0 para poder realizar un movimiento

MASCARA DE BORRADO:

		1					
1					1		
	1		1			1	
1				1	1		1
		1				1	
			1				
1					1		

APLICACIÓN DE DECISIONES TOMADAS

CAMBIOS EN TABLERO:

4	16		8		2		
	4	4	4	8			
4		4		4	8		8
	16	8	4			8	
4	8				8		4
16	2					8	8
	8	8		16		8	
16	4	16	8		8	2	

Se realizan los cambios correspondientes borrando donde la matriz de borrado indica y poniendo una ficha donde la de decisiones lo hace

CENTA DE CEROS, MATRIZ DE SALTOS

Se cuenta la cantidad de 0s que hay desde cada casilla el final de su fila o columna.

POSICIONES A DESPLAZAR CADA ELEMENTO:

		1		1		1	1
1		1		1	1	2	2
1	1	1	1	1	1	3	2
2	1	1	1	2	2	3	3
2	1	2	2	3	2	4	3
2	1	3	3	4	3	4	3
3	1	3	4	4	4	4	4

SALTO

Se mueven las casillas del tablero tanto como indique la matriz de salto.

ELEMENTOS DESPLAZADO:

4	16	4	8	8	2	8	8
4	4	4	4	4	8	8	4
4	16	8	4	16	8	8	8
16	8	8	8		8	2	
16	2	16					
	8						
	4						

(Los elementos que estaban a 0 no se desplazan)

RE-FLIP

Se vuelve a hacer el espejo de la matriz por el eje correspondiente para obtener el resultado final del movimiento

MATRIZ FINAL:

	4						
	8						
16	2	16					
16	8	8	8		8	2	
4	16	8	4	16	8	8	8
4	4	4	4	4	8	8	4
4	16	4	8	8	2	8	8

CUENTA DE MOVIMIENTOS

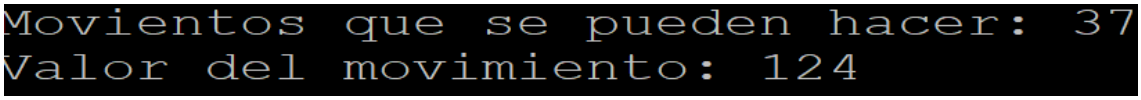
Se genera una matriz con un 1 en cada posición en la que se pueda hacer un movimiento

Sumando el resultado de todos los unos sabremos si se pueden hacer movimientos o no.

Cada hilo mira a sus cuatro elementos de los lados y al suyo.

Se utiliza para saber cuando ya no quedan movimientos y por tanto terminar la partida o quitar una vida según corresponda.

Huecos: 9



KERNEL DE SUMA

Hay distintas versiones: suma la cantidad de 0s, suma la cantidad de valores distintos de 0 o suma toda la matriz.

Suma 0s: se utiliza para saber si hay huecos en la matriz.

Suma distintos de 0: se utiliza para la heurística de la IA, para valorar los buenos que son sus movimientos. Y para saber si una matriz ha cambiado o no.

Todas ellas aplican reducción para sumar matrices.

COMPROBACIÓN DE MOVIMIENTO VÁLIDO

Se comprueba si el tablero anterior es igual al nievo tras el movimiento de modo que no se suman los puntos si este no ha producido cambios.

3. Límites y Dificultades

La mayor dificultad técnica a la hora de hacer la práctica ha venido del paso de global a bloques. En global toda la matriz está siempre en un único bloque lo que permite la sincronización de hilos. No obstante, al pasar a bloques donde habrá hilos en varios de ellos esto ya no es posible. Para solventarlo nos hemos dado cuenta de la utilidad de tener unas matrices desde las que solo se va a leer y otras en las que solo se va a escribir como parámetros de cada kernel. Leer y escribir en la misma matriz puede dar problemas debido a la sincronización de hilos.

Otra opción para resolver esto ha sido realizar múltiples llamadas a los kernell como en el caso de las sumas por reducción. Un parámetro de kernel es un índice que indica los valores que se deben mirar para hacer la suma.

4. Partes realizadas

Se ha realizado memoria por bloque con teselado. Para ello se utiliza la tesela que gaste menos hilos la cual es calculada para cada partida según la gráfica en la que se vaya a ejecutar.

También se ha realizado memoria compartida, aunque parcialmente, no se ha aplicado en todos los kernel debido a las dificultades encontradas.

No se ha realizado interfaz gráfica con GLUT aunque la interfaz de terminal es bastante cuidada.

Por el resto se cumplen todos los demás requisitos de la práctica como llevar la cuenta de los puntos o cargar y guardar matrices.

5. Inteligencia Artificial

Hay dos modos de IA, una IA aleatoria que realiza movimientos dando favoritismo a dos de ellos de modo que busca concentrar estos en una posición concreta.

La otra versión evalúa los 4 movimientos posibles y realiza aquél que tenga más puntuación basada en la siguiente heurística.

Nunca realizar un movimiento que acabe la partida si hay otero que no lo hace.

Preferir movimientos que cambien muchas posiciones de sitio.

Preferir movimientos que proporcionen más puntuación.

6. Funciones de la GPU

6.1.Void flipH (float *tablero, int size, int nc)

Realiza un volteo de la matriz, convirtiéndola en su simétrica, respecto al eje vertical. Con esto implementamos el movimiento hacia la derecha.

Para ello se toman los diferentes hilos, calculando el número de columna y fila que le corresponde para hacer su simétrica y se realiza el cambio.

Se comprueba que tanto el valor de su simétrica como el hilo no sean mayores que el tamaño de la matriz.

6.2.Void flipV(float *tablero, int size, int nc)

Del mismo modo que con la función anterior, esta implementa el movimiento hacia abajo y se realiza su simétrica a través del eje horizontal.

Sigue el mismo proceso que la función anterior.

6.3.void createDeleterV(float *tablero, float *decisions, float *out, int size, int nc)

Con esta función se crea la máscara de borrado, para eliminar todos los números que se han juntado.

6.4.void moveH(float *tablero, float *decisions, int size, int nc)

Se toma como entrada la matriz de tablero actualizada (tras el borrado e inclusión de la toma de decisiones) y la matriz con los saltos que debería hacer cada casilla para eliminar los 0s del tablero en el sentido del movimiento

6.5.void moveV(float *tablero, float *decisions, int size, int nc, int nf)

Igual que la anterior pero para la dirección veritcal.

6.6.Void takeDecisionsH(float *tablero, float *decisions,int size, int nc)

Decide van a ir los números del próximo resultado. Además, sirve para contar los puntos de la partida pues su resultado es una matriz con las nuevas piezas que aparecen al hacer el movimiento.

6.7.Void sumLeft(float* tablero, float* result,int size, int nc, int nf)

Se analiza si se pueden realizar movimientos.

Para ello, tomamos el cociente de dividir el identificador del hilo entre el número de columnas y el resto de la misma operación en las variables, columna y fila. Controlando que el hilo no se salga del tamaño de la matriz, se pone a 0 el array de resultados. A continuación, comprobamos a la izquierda, arriba, abajo y a la derecha si los elementos son iguales. De ser así, marcamos en el array de resultados, a 1 esa posición.

6.8.Void sumPoints(float *decisions, float *sum_result, int size, int max_steps)

Suma el número de puntos que se consiguen con un movimiento. Para ello, en el array de sum_result ponemos el valor de la posición que ocupa el hilo en el array decisiones. Después mediante el algoritmo de reducción binaria, se van sumando los pares hasta llegar a un único

6.9. Void sumGaps(float *decisions, float *sum_result, int size, int max_steps.

Se cuentan los huecos libres que hay en la matriz. Tiene la misma implementación que la función anterior.

6.10. Void sumMovements(float *decisions, float *sum_result, int size, int max_steps)

Al igual que las anteriores funciones sigue la misma metodología. En este caso, se usa para sumar la cantidad de cambios producidos en el tablero lo que sirve para saber si el movimiento es válido y también como heurística de IA.

6.11. Void cpyMatrix(float *matriz, float *copia, int size)

Esta función es una función auxiliar, que permite copiar una matriz en otra.

6.12. Void hasChanged(float *matriz, float *copia, int size)

Otra función auxiliar para comprobar si ha cambiado el tablero. De ser así, lo copiamos en la segunda matriz.

7. Funciones de la CPU

7.1. Void addRandom (T *tablero, int elements, int len)

Con esta función lanzada dese la CPU metemos la cantidad de números aleatorios correspondientes. Para ello se reserva una posición de memoria. Después, a través de un bucle, si la posición esta vacía se añade a las posiciones disponibles.

En el caso de que no haya posiciones libres se sale de la función. Y por último, de forma aleatoria, dentro de todas las posiciones libres que quedan, se escoge una y se introduce un 4 o un 2.

7.2. sumArray(T *arr, int len)

Suma la puntuación de las 5 vidas.

7.3. T maxArray(T *arr, int len)

Calcula el máximo de puntos en modo automático, de un array de 4 posiciones.

7.4. std::string printTablero(T *tablero, int n_columnas, int n_filas)

Con esta función ejecutada en la CPU, procedemos a mostrar el tablero a través del terminal. Cada número tiene un color asociado. Su funcionamiento es un bucle que muestra todo el tablero.

8. Main

En el main del programa, se declaran todas las variables relacionadas con el host y con el device que van a ser necesarias a lo largo de la ejecución del programa. Además, se sigue el proceso básico de reserva de memoria tanto en CPU y GPU y sus consiguientes envíos de información del host al device y viceversa.

Dentro del main, también encontramos las posibilidades de juego que tiene que introducir el usuario por teclado.

Se hacen comprobaciones de los datos introducidos para que el tablero sea correcto. Y por último se controlan los errores de la GPU con su mensaje de aviso.