MEMORIA PECL1 APA

Sanz Sacristán, Sergio
Zamorano Ortega, Álvaro

Índice

MEMORIA GLOBAL 1 BLOQUE

FUNCIONES

Para esta parte de la práctica hemos realizado una serie de distintas funciones las cuales vamos a explicar a continuación.

Función main()

En esta función realizamos los pasos previos y llamamos a las funciones necesarias para la inicialización del programa, es decir, llamamos a las funciones correspondientes a recoger los datos con los que trabajará el programa del usuario, y las funciones necesarias para la generación de la matriz que corresponderá con el tablero con el que se va a jugar.

Primero preguntamos si queremos cargar una partida guardada en caso de que tengamos alguna.

```
49      printf("Quieres cargar una partida anterior o empezar de nuevo ? (s: si | n : no)\n");
50      fflush(stdin);
51      scanf("%c", &cargarP);
```

En caso de elegir la opción no, inicializamos el número de vidas de las que vamos a disponer durante el juego, que inicialmente serán 5. También inicializamos el número de puntos contabilizados en la partida que se iniciará en 0.

```
58      if (cargarP == 'n') {
59
60          int vida = 5;
61          int *numVidas;
62          numVidas = &vida;
63
64          int numPuntos = 0;
65          int *puntos;
66          puntos = &numPuntos;
```

```
68          modo = pedirModo();
69          dificultad = pedirDificultad();
70          filas = pedirFilasTablero();
71          columnas = pedirColumnasTablero();
```

Luego pedimos el modo de juego, es decir automático o manual, para ello llamamos a la función pedirModo(). La dificultad del juego, fácil o difícil, a través de la función pedirDificultad(). Pedimos al usuario el número de filas que va a tener el tablero de su partida a través de la función pedirFilasTablero(). Por último, pedimos el número de columnas que va a tener el tablero del juego a través de la función pedirColumnasTablero(). Todas estas funciones las explicaremos un poco más en profundidad más adelante.

```
80          matriz = generarMatriz();
81          rellenarMatriz(matriz);
82          cargarRecord();
```

Posteriormente generamos la matriz llamando a la función generarMatriz() siguiendo las medidas de tablero introducidas por el usuario y luego la rellenamos dependiendo del modo de dificultad seleccionado. Estas matrices las explicaremos posteriormente.

Por último, llamamos a la función jugar() pasándola como atributos la matriz, el número de vidas y los puntos de la partida. Esta función será la que lleve la mayor parte del peso del proceso llevado a cabo durante el juego. También hay que destacar si contestamos a cargar partida no realizará todo este proceso y llamará directamente a la función cargarPartida().

Función comprobarPropiedades()

```
if ((filas*columnas * sizeof(int)) >= globalMem) {

    printf("La matriz solicitada ocupa %zd y excede la capacidad de memoria global de tu tarjeta gráfica que es %zd \n",
        filas*columnas * sizeof(int), globalMem);
    error = 'T';
}
if ((filas*columnas > hilosPorBloque)) {

    printf("La matriz solicitada ocupa %d hilos y excede la cantidad de hilos por bloque que es%zd \n",
        filas*columnas, hilosPorBloque);
    error = 'T';
}
```

Esta función se encarga de consultar las características de la tarjeta grafica del ordenador donde se ejecuta el programa. En caso de que las dimensiones de la matriz sean superiores al límite de la tarjeta gráfica o al límite de la cantidad de hilos por bloque (en el caso de memoria global a 1 bloque) nos imprimirá un mensaje de error.

Función generarMatriz()

```
148  int *generarMatriz() {
149      int* matriz = (int*)malloc(filas*columnas * sizeof(int));
150
151      for (int i = 0; i < filas*columnas; i++) {
152          matriz[i] = 0;
153      }
154      return matriz;
155  }
```

En esta función generamos una matriz con el número de filas y columnas introducidos por el usuario y la inicializa a 0 en todas sus posiciones.

Función rellenarMatriz()

```
157  bool rellenarMatriz(int* matriz) {
158      bool terminado = false;
159      int numSemillas;
160      int numAleatorio;
161      int random;
162      time_t t;
```

En esta función rellenamos la matriz creada previamente, dependiendo de la dificultad seleccionada al principio del juego esta función rellenará la matriz con 15 semillas, con valores 2, 4 u 8 si está en dificultad fácil y 8 semillas con valores 2 o 4 si esta en dificultad difícil. Lo que hace es recorrer la matriz y de forma aleatoria introducir el número de semillas correspondiente con uno de los valores posibles. En el caso de que en la posición aleatorio haya ya un número distinto de 0, se volverá a hacer bucle. Adjuntamos imagen con este proceso en caso de haber seleccionado nivel de dificultad fácil.

```
166     if (dificultad == 'F') {
167         numSemillas = 15;
168         if (recuento(matriz) < numSemillas) {
169             terminado = true;
170         }
171         else {
172             int posiblesNum[] = { 2, 4,8 };
173             numAleatorio = 3;
174             while (numSemillas > 0 && !terminado) {
175                 random = rand() % (filas*columnas);
176                 if (matriz[random] == 0) {
177                     matriz[random] = posiblesNum[rand() % numAleatorio];
178                     numSemillas = numSemillas - 1;
179                 }
180             }
181             if (recuento(matriz) < 15) {
182                 terminado = true;
183             }
184         }
185     }
```

Función recuento()

```
209 int recuento(int* matriz) {
210     int recuento = 0;
211     for (int i = 0; i < filas*columnas; i++) {
212         if (matriz[i] == 0) {
213             recuento = recuento + 1;
214         }
215     }
216     return recuento;
217 }
```

Esta función se utiliza en la función anterior. Esta función se utiliza para contabilizar el número de 0s y así saber si se pueden añadir el número de semillas correspondiente al nivel de dificultad, y si no se puede, acabará en “GAME OVER”.

Función pedirModo()

```
220 char pedirModo() {
221     char modo = ' ';
222     getchar();
223     while (modo != 'M' && modo != 'A') {
224         printf("Que modo desea para el juego? Automatico (A), Manual (M)\n");
225         fflush(stdin);
226         scanf("%c", &modo);
227         if (modo != 'M' && modo != 'A') {
228             printf("Usted ha introducido un modo que no existe: -%c.\n", modo);
229             printf("Por favor, introduzca uno de los siguientes dmodos que se le presentan por pantalla.\n\n");
230             scanf("%c", &modo);
231         }
232     }
233     return modo;
234 }
```

Esta función es la encargada de pedir al usuario que modo de juego se va a llevar a cabo en el juego, se presentan dos opciones o de forma automática, en la cual los movimientos los realizará de forma automática, es decir, el jugador será el ordenador, o el modo manual en el cual el usuario introduce a través de las flechas los movimientos que se realizarán en el juego.

Función pedirDificultad()

```
237 char pedirDificultad() {
238     char dificultad = ' ';
239     getchar();
240     while (dificultad != 'F' && dificultad != 'D') {
241         printf("Que dificultad desea para el juego? Facil (F), Dificil (D)\n");
242         fflush(stdin);
243         scanf("%c", &dificultad);
244         if (dificultad != 'F' && dificultad != 'D') {
245             printf("Usted ha introducido una dificutad que no existe: -%c.\n", dificultad);
246             printf("Por favor, introduzca uno de las siguientes dificultades que se le presentan por pantalla.\n\n");
247             scanf("%c", &dificultad);
248         }
249     }
250     return dificultad;
251 }
```

Esta función es la encargada de pedir al usuario con que dificultad desea jugar, hay dos opciones, o en nivel fácil o en nivel difícil. Esta función tan solo pide por pantalla este dato y lo guarda en el atributo correspondiente, nombrado en la función main().

Función pedirFilasTablero()

```
254 int pedirFilasTablero() {
255     cudaDeviceProp prop;
256     cudaGetDeviceProperties(&prop, 0);
257     int tamMaximo = prop.maxThreadsPerBlock;
258     int filas;
259     do {
260         printf("\nIntroduzca las filas que tendra el tablero: ");
261         fflush(stdin);
262         scanf("%d", &filas);
263         if (filas < 1 || filas > tamMaximo) {
264             printf("Introduzca un numero de filas correcto\n");
265         }
266     } while (filas < 1); //El numero de filas tiene que ser un numero entero positivo
267
268     return filas;
269 }
```

Esta función es la encargada de preguntar al usuario cual va a ser el número de filas del tablero con el que se va a realizar la partida. En el caso de memoria global a 1 bloque, no se podrán introducir un número de filas mayor al número máximo de hilos limitados por bloque en la tarjeta gráfica.

Función pedirColumnasTablero()

```
272 int pedirColumnasTablero() {
273     cudaDeviceProp prop;
274     cudaGetDeviceProperties(&prop, 0);
275     int tamMaximo = prop.maxThreadsPerBlock;
276     int columnas;
277
278     do {
279         printf("\nIntroduzca las columnas que tendra el tablero: ");
280         fflush(stdin);
281         scanf("%d", &columnas);
282         if (columnas < 1 || columnas > tamMaximo) {
283             printf("Introduzca un numero de columnas correcto\n");
284         }
285     } while (columnas < 1); //El numero de filas tiene que ser un numero entero positivo
286
287     return columnas;
288 }
```

Esta función realiza el mismo proceso que la función anterior, pero pidiendo al usuario el número de columnas que forman el tablero de juego. En el caso de memoria global a 1 bloque, no se podrán introducir un número de columnas mayor al número máximo de hilos limitados por bloque en la tarjeta gráfica.

Función imprimirMatriz() e imprimirEspacios()

Estas funciones son las encargadas de imprimir la matriz de tal forma que se ajusten cada una de las columnas entre si para así imprimir una matriz simétrica y bien definida para que sea posible seguir el proceso de los movimientos y las sumas de dentro de la matriz.

Función sumaPuntAux()

```
751  int sumaPuntAux(int tamaño, int* punt) {
752      int suma = 0;
753      for (int i = 0; i < tamaño; i++) {
754          suma += punt[i];
755      }
756      return suma;
757  }
```

Esta función se encarga de sumar de forma parcial el valor de cada una de las celdas sumadas en cada movimiento de los valores de la matriz.

Función jugar()

```
760  void jugar(int *matriz, int* numVidas, int* puntos) {
761      int *dev_matriz;
762      int *dev_puntos;
763      char movimiento = ' ';
764      int numRan;
765      bool terminado;
766      int* puntos_aux = generarMatriz();
```

En esta función gestionamos el programa durante el juego. Mientras el número de vidas sea mayor que 0 nos mantenemos en un bucle que estará activo hasta que se agoten las vidas, y, por lo tanto, finalice la partida. Si nos encontramos en una partida con el modo de juego manual, con la función _getch() gestionamos que teclas pulsa el usuario y si corresponde con alguna de las flechas queda registrado y si pulsa la "g" llama a guardarPartida() para guardar la partida en ese momento. Los movimientos los identificamos con los caracteres 'W' (arriba), 'A' (izquierda), 'S'(abajo) y 'D' (derecha). Si se pulsa la tecla 'g', se llamará a la función guardarPartida() y se deberán seguir pulsando las flechas para jugar.

```
775             if (modo == 'M') {
776                 printf("Pulse una flecha...Pulse g para guardar");
777                 bool bucle = true;
778 
779                 while (bucle) {
780                     movimiento = _getch();
781                     switch (movimiento) {
782                     case 72:
783                         movimiento = 'W'; //Arriba
784                         bucle = false;
785                         break;
786                     case 80:
787                         movimiento = 'S'; //Abajo
788                         bucle = false;
789                         break;
790                     case 75:
791                         movimiento = 'A'; //Izquierda
792                         bucle = false;
793                         break;
794                     case 77:
795                         movimiento = 'D'; //Derecha
796                         bucle = false;
797                         break;
798                     case 103:
799                         //SI PULSAS G GUARDA LA PARTIDA
800                         guardarPartida(matriz, numVidas, puntos);
801                         break;
802                     }
803                 }
804             }
```

En cambio, si el modo de juego en automático entonces seleccionamos uno de los cuatro movimientos posibles de forma aleatoria para realizar el movimiento.

```
805             else {
806                 numRan = rand() % 4;
807                 switch (numRan) {
808                 case 0:
809                     movimiento = 'W';
810                     break;
811                 case 1:
812                     movimiento = 'A';
813                     break;
814                 case 2:
815                     movimiento = 'S';
816                     break;
817                 case 3:
818                     movimiento = 'D';
819                     break;
820                 }
821             }
```

Posteriormente, reservamos memoria con cudaMalloc para la matriz y los puntos obtenidos en el juego y copiamos los datos correspondientes del host al device.

Luego mediante un switch, dependiendo del movimiento a realizar captado por las operaciones anteriores, llamamos a los kernels correspondientes, en este caso al estar en memoria global a un bloque, el número de bloques que se ejecutarán será 1 y el número de hilos que se ejecutarán en el bloque será el correspondiente a la operación filas * columnas (los kernels los explicaremos más adelante).

```
838             switch (movimiento) {
839             case 'W':
840                 printf("\nARRIBA");
841                 sumarElementosArriba << < 1, filas*columnas >> > (dev_puntos, dev_matriz, columnas);
842                 moverElementosArriba << < 1, filas*columnas >> > (dev_matriz, filas, columnas);
843                 break;
844             case 'A':
845                 printf("\nIZQUIERDA");
846                 sumarElementosIzquierda << < 1, filas*columnas >> > (dev_puntos, dev_matriz, columnas);
847                 moverElementosIzquierda << < 1, filas*columnas >> > (dev_matriz, filas, columnas);
848                 break;
849             case 'S':
850                 printf("\nABAJO");
851                 sumarElementosAbajo << < 1, filas*columnas >> > (dev_puntos, dev_matriz, filas, columnas);
852                 moverElementosAbajo << < 1, filas*columnas >> > (dev_matriz, filas, columnas);
853                 break;
854             case 'D':
855                 printf("\nDERECHA");
856                 sumarElementosDerecha << < 1, filas*columnas >> > (dev_puntos, dev_matriz, columnas);
857                 moverElementosDerecha << < 1, filas*columnas >> > (dev_matriz, filas, columnas);
858                 break;
859             }
```

Para finalizar recuperamos los datos de la matriz y los puntos del device y comprobamos si el número de puntos conseguidos es mayor al récord, para establecer el nuevo récord en ese caso. Rellenamos la matriz con el número de semillas correspondiente y la imprimimos. Si al imprimirla nos devuelve un booleano a false, significará el fin de la partida dado que no se puede rellenar la matriz con el número de semillas de la dificultad dada, por lo que se le restará a 1 el número de vidas y se volverá a empezar una nueva partida con otro tablero.

Funciones cargar y guardar partida

Disponemos de dos funciones cargar y dos guardar; dos de ellas se encargan de guardar y cargar partida en un archivo txt llamado “guardado.txt” y otros dos de guardar y cargar el récord a lo largo de las partidas en un archivo txt llamado “record.txt”. En estas funciones se guardan los datos a través de fprintf y se cargan a través de fscanf. Adjuntamos imagen de las dos funciones de guardar y cargar récord.

```
953 void cargarRecord() {
954
955     leer = fopen("record.txt", "r");
956
957     fscanf(leer, "%d", &punt_record);
958 }
```

```
960 void guardarRecord() {
961     doc = fopen("record.txt", "w");
962     fprintf(doc, "%i \n", punt_record);
963     fclose(doc);
964     printf("\n--GUARDADO RECORD--\n");
965 }
```

KERNELS

Tenemos tantos kernels como posibilidad de movimientos para sumar, y tantos kernels como posibilidad de movimientos, para mover. Es decir, disponemos de 4 kernels de suma, cuyo objetivo es sumar en función del movimiento que sigue si los valores de la misma fila/columna son iguales. Y otros 4 kernels que realizan el movimiento correspondiente después de realizar la suma.

En todos los kernels pasamos como atributos los puntos, la matriz generada y el número de columnas introducido por el usuario. A los kernels de movimiento se les pasa la matriz cambiada por los kernels de suma (con las posiciones sumadas).

sumarElementosDerecha()

La posición correspondiente de cada uno de los hilos es identificada por threadIdx.x. De esta forma realizamos un bucle con do while (que recorrerá la fila correspondiente al hilo y sumará tantos elementos como coincidentes sean con el valor del hilo actual), el cual se ejecutará hasta que la variable terminado pase a ser true. Si la posición se encuentra en la última columna de la

derecha (el límite), o en la posición en la que se encuentra el hilo es de valor 0, o según recorre la fila correspondiente al hilo se encuentra con un número que no sea 0 y no sea igual al valor del hilo en el que estás, entonces se pone terminado a true y sale del bucle. Si durante este bucle encontramos un valor igual al del hilo actual, sumaremos la variable numElementos a 1. Al salir del bucle, si la variable numElementos es par, se sumará el hilo a la posición que haya coincidido primero (variable primero). También incluimos __syncthreads() para esperar a todos los hilos del bloque después de recorrer la fila de cada hilo y después de realizar la suma en el caso de que sea necesario.

```
do {
    if ((pos + 1) % numColumnas == 0 || matriz[pos] == 0 || (matriz[pos + i] != 0 && matriz[pos + i] != matriz[pos])) {
        terminado = true;
    }
    else {
        if (matriz[pos] == matriz[pos + i]) {
            if (primero) {
                posElementoSuma = pos + i;
                primero = false;
            }
            numElementos = numElementos + 1;
        }
        if ((pos + 1 + i) % numColumnas == 0) {
            terminado = true;
        }
        i++;
    }
} while (terminado == false);
```

```
if (numElementos % 2 == 0) {
    suma = true;
}

__syncthreads();

if (suma) {
    matriz[posElementoSuma] = matriz[posElementoSuma] + matriz[pos];
    matriz[pos] = 0;
    puntos[pos] = matriz[posElementoSuma];
}
__syncthreads();
```

sumarElementosIzquierda()

En este kernel realizamos casi de forma similar la suma de los valores de dentro de la matriz, pero recorriendo la fila de cada hilo hacia la izquierda. Ahora el límite que debe reconocer es de la primera columna de la izquierda de la matriz.

```
do {
    if (pos % numColumnas == 0 || matriz[pos] == 0 || (matriz[pos - i] != 0 && matriz[pos - i] != matriz[pos])) {
        terminado = true;
    }
    else {
        if (matriz[pos] == matriz[pos - i]) {
            if (primero) {
                posElementoSuma = pos - i;
                primero = false;
            }
            numElementos = numElementos + 1;
        }
        if ((pos - i) % numColumnas == 0) {
            terminado = true;
        }
        i++;
    }
} while (terminado == false);
```

```
if (suma) {
    matriz[posElementoSuma] = matriz[posElementoSuma] + matriz[pos];
    matriz[pos] = 0;
    puntos[pos] = matriz[posElementoSuma];
}
```

sumarElementosArriba()

En este caso realizamos el mismo proceso, pero ahora la i, para recorrer la columna correspondiente al hilo en el que nos encontramos, en vez de incrementarse de uno en uno se incrementa según el número de columnas de la matriz. En este bucle debemos definir el límite en la primera fila de la matriz.

```
do {
    if (pos < numColumnas || matriz[pos] == 0 || (matriz[pos - i] != 0 && matriz[pos - i] != matriz[pos])) {
        terminado = true;
    }
    else {
        if (matriz[pos] == matriz[pos - i]) {
            if (primero) {
                posElementoSuma = pos - i;
                primero = false;
            }
            numElementos = numElementos + 1;
        }
        if ((pos - i) < numColumnas) {
            terminado = true;
        }
        i = i + numColumnas;
    }
} while (terminado == false);
```

```
if (suma) {
    matriz[posElementoSuma] = matriz[posElementoSuma] + matriz[pos];
    matriz[pos] = 0;
    puntos[pos] = matriz[posElementoSuma];
}
```

sumarElementosAbajo()

En este caso comprobaremos si tenemos que realizar la suma recorriendo la columna correspondiente al hilo en el que nos encontramos hacia abajo con la i de la misma forma que en el kernel anterior. En este caso, el límite será la última fila de la matriz.

```
do {
    if (pos >= (numColumnas*(numFilas - 1)) || matriz[pos] == 0 || (matriz[pos + i] != 0 && matriz[pos + i] != matriz[pos])) {
        terminado = true;
    }
    else {
        if (matriz[pos] == matriz[pos + i]) {
            if (primero) {
                posElementoSuma = pos + i;
                primero = false;
            }
            numElementos = numElementos + 1;
        }
        if ((pos + i) >= (numColumnas*(numFilas - 1))) {
            terminado = true;
        }
        i = i + numColumnas;
    }
} while (terminado == false);
```

```
if (suma) {
    matriz[posElementoSuma] = matriz[posElementoSuma] + matriz[pos];
    matriz[pos] = 0;
    puntos[pos] = matriz[posElementoSuma];
}
```

moverElementosDerecha()

Mediante la llamada a este kernel, los elementos de la matriz generada por el kernel de sumarElementosDerecha() se desplazarán hacia la derecha del tablero. Los hilos son identificados mediante threadIdx.x, por lo que de esta forma sabremos la posición de la matriz a la que tienen que mover en el tablero. El funcionamiento se realiza mediante un bucle do while: en la primera condición estudiamos si la posición en la que se encuentra el hilo dentro del tablero está en la columna de la derecha límite(si lo está se saldrá del bucle y esa posición no se moverá); si no lo está, se sumarán el número de ceros de las posiciones contiguas hacia la derecha (mediante un índice, el cual se inicializará a 1, que se sumará en cada bucle del while) hasta llegar a la columna límite, momento en el que se saldrá del bucle.

Si se ha encontrado algún cero, existirá movimiento, por lo que el hilo desplazará la posición con tantos ceros se hayan sumado en el bucle.

```
do {
    if ((pos + 1) % numColumnas == 0) {
        terminado = true;
    }
    else {
        if (matriz[pos] == 0) {
            terminado = true;
        }
        else {
            if (matriz[pos + i] == 0) {
                numElementos = numElementos + 1;
            }

            if ((pos + i + 1) % numColumnas == 0) {
                if (numElementos > 0) {
                    mov = true;
                }
                terminado = true;
            }
        }
        i++;
    }
} while (terminado == false);
```

```
__syncthreads();

if (mov) {
    matriz[pos] = 0;

}
__syncthreads();
if (mov) {
    matriz[pos + numElementos] = valor;
}
```

moverElementosIzquierda()

El funcionamiento del bucle de este kernel es similar al anterior, lo único que cambia es el establecimiento del límite (primera columna) y el movimiento, que se realizará restando el número de ceros sumados de esa fila.

```
do {
    if (pos % numColumnas == 0) {
        terminado = true;
    }
    else {
        if (matriz[pos] == 0) {
            terminado = true;
        }
        else {
            if (matriz[pos - i] == 0) {
                numElementos = numElementos + 1;
            }

            if ((pos - i) % numColumnas == 0) {
                if (numElementos > 0) {
                    mov = true;
                }
                terminado = true;
            }
        }
        i++;
    }
} while (terminado == false);
```

```
__syncthreads();

if (mov) {
    matriz[pos] = 0;
}
__syncthreads();
if (mov) {
    matriz[pos - numElementos] = valor;
}
```

moverElementosArriba()

La diferencia respecto a los anteriores kernel de movimiento son las siguientes: el límite será la primera fila, el índice que recorre la columna será el número de columnas del tablero y que se sumará en esa misma cantidad cada vez que recorre el bucle, y finalmente, los movimientos, que se realizarán restando a la posición inicial el número de 0s encontrados multiplicado por el número de columnas.

```
do {
    if (pos < numColumnas) {
        terminado = true;
    }
    else {
        if (matriz[pos] == 0) {
            terminado = true;
        }
        else {
            if (matriz[pos - i] == 0) {
                numElementos = numElementos + 1;
            }

            if ((pos - i) < numColumnas) {
                if (numElementos > 0) {
                    mov = true;
                }
                terminado = true;
            }
        }
        i = i + numColumnas;
    }
} while (terminado == false);
```

```
__syncthreads();

if (mov) {
    matriz[pos] = 0;
}
__syncthreads();
if (mov) {
    matriz[pos - (numElementos * numColumnas)] = valor;
}
```

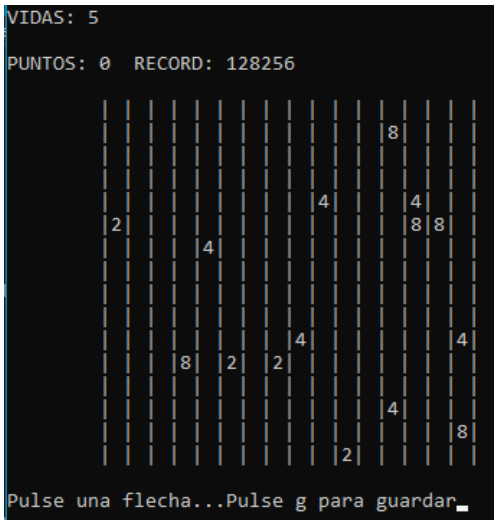
moverElementosAbajo()

Respecto al kernel de movimiento hacia abajo, se diferencia en: la identificación del límite, que será la última fila del tablero, y los movimientos, que se realizarán sumando a la posición inicial el número de 0s encontrados multiplicado por el número de columnas.

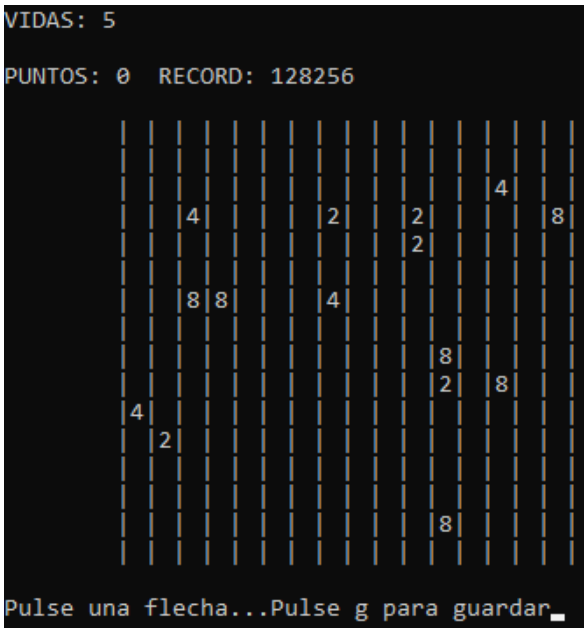
```
do {
    if (pos >= numColumnas * (numFilas - 1)) {
        terminado = true;
    }
    else {
        if (matriz[pos] == 0) {
            terminado = true;
        }
        else {
            if (matriz[pos + i] == 0) {
                numElementos = numElementos + 1;
            }

            if ((pos + i) >= numColumnas * (numFilas - 1)) {
                if (numElementos > 0) {
                    mov = true;
                }
                terminado = true;
            }
        }
        i = i + numColumnas;
    }
} while (terminado == false);
```

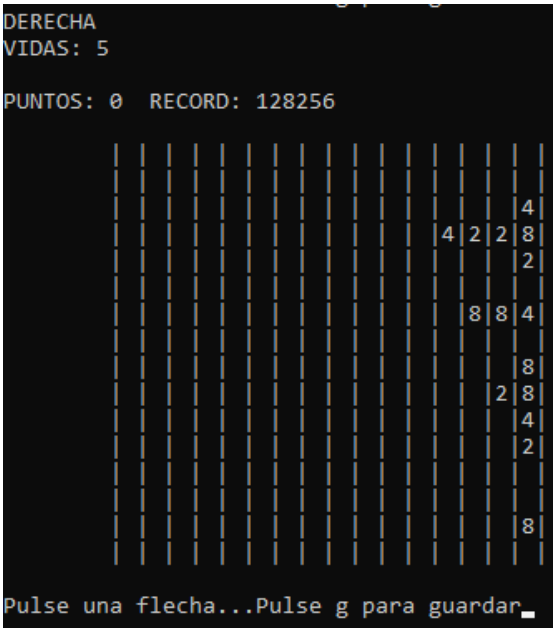
```
if (mov) {
    matriz[pos] = 0;
    __syncthreads();
}
if (mov) {
    matriz[pos + (numElementos * numColumnas)] = valor;
}
```

Ejemplo de suma sin insertar semillas

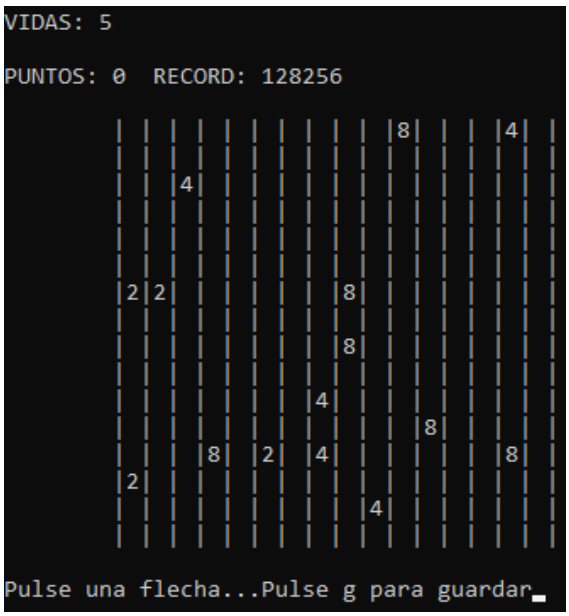


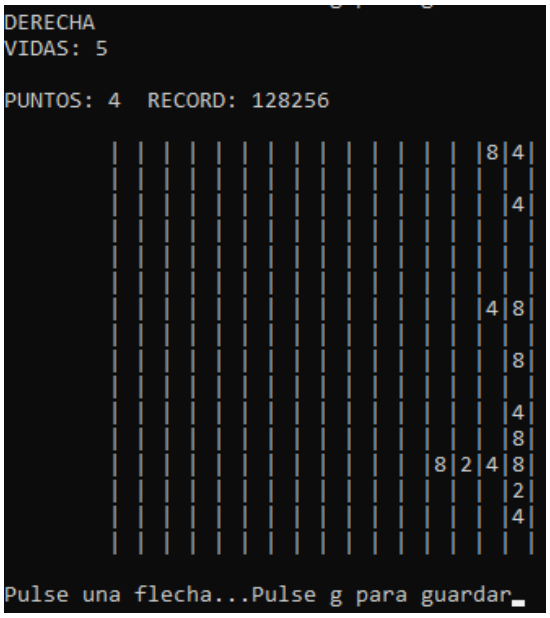
Ejemplo de movimiento sin insertar semillas





Ejemplo de suma y movimiento sin insertar semillas





MEMORIA GLOBAL MULTIBLOQUE

Las diferencias existentes con respecto al programa realizado a un bloque son:

Función jugar()

Hemos introducido dos nuevas matrices auxiliares:

- matriz_aux: matriz auxiliar que almacena temporalmente los movimientos de los elementos del tablero. Esta matriz es necesaria ya que, si los movimientos de los elementos los introducíamos directamente, había ocasiones que se solapaban debido a que los hilos de un bloque no veían a los de los otros y no se esperaban a que terminaran de ejecutarse.
- matriz_suma: matriz auxiliar que almacena temporalmente las posiciones sumadas de los elementos del tablero cuando se produce movimiento. Al igual que lo explicado en la matriz anterior, al no cooperar los hilos de diferentes bloques, se producían errores al sumar si se introducían los elementos ya sumados directamente en la matriz.
- Variable hilospBloque: esta variable de tipo entero representa el número de hilos por bloque. Esta variable llama al método hilosBloque (int size), que devuelve el número de hilos óptimo que debe haber para que se desperdicien el menor número posible de hilos que se encuentran fuera de los límites del tablero. Para calcular su número, se calcula el mínimo común múltiplo entre 3 teselas de 64,256 y 1024 hilos (teselas 8,16 y 32) y el número de filas por columnas, y la que de el menor número será la tesela a utilizar.
- Variable tesela: representa el ancho que tendrán los bloques que divide a la matriz. Esta se calculará como la raíz cuadrada de la variable hilospBloque.
- dimGrid y dimBlock:

```
dim3 dimGrid(columnas + tesela - 1 / tesela, filas + tesela - 1 / tesela);//Tamaño del grid en bloques
dim3 dimBlock(tesela, tesela);//Tamaño de los bloques en hilos
```

Posteriormente a la llamada de ejecución de los kernel, se cambian los elementos de la matriz por los elementos de la matriz auxiliar:

```
for (int k = 0; k < filas*columnas; k++) {//Almacenamos en el tablero los elementos de la matriz auxiliar
    matriz[k] = matriz_aux[k];
}
```

Kernel de sumas y de movimientos

- Ahora, los hilos se representan por su fila y por su columna dentro del tablero:

```
int columna = threadIdx.x + blockIdx.x * blockDim.x;
int fila = threadIdx.y + blockIdx.y * blockDim.y;
```

```
int pos = fila * numColumnas + columna;
```

- Debido a que hora se realiza por más de un bloque, existirán hilos que no deberán ejecutar el bucle, por lo que al principio introducimos una condición que deseche aquellos hilos que están fuera de los límites:

```
if (fila < numFilas && columna < numColumnas) {//Si la fila y la columna estan dentro de los limites
```

- Como hemos explicado anteriormente, ahora se almacenan las posiciones sumadas en una matriz auxiliar:

```
//Si el valor de la celda es distinto de 0, no suma y la posicion en la auxiliar es 0
if (valor != 0 && !suma  && matriz_suma[pos] == 0) {
    matriz_suma[pos] = valor;
}

//Realiza la suma, la almacena en la matriz_suma y contabiliza los puntos
if (suma) {
    matriz_suma[posElementoSuma] = matriz[posElementoSuma] + matriz[pos];//Se mete el valor a la matriz suma auxiliar
    puntos[pos] = matriz_suma[posElementoSuma];
}
```

- Los movimientos ahora se guardan en una matriz auxiliar:

```
if (valor != 0 && !mov) {//Si el valor de la posicion del hilo es distinto de 0 y no se mueve
    matriz_aux[pos] = valor;
}

if (mov) {
    matriz_aux[pos + (numElementos * numColumnas)] = valor;//Se almacena el elemento en la matriz auxiliar
}
```