




15 DE MARZO DE 2019

16384: PECL1 APA

UNIVERSIDAD DE ALCALÁ DE HENARES

LUIS ALEJANDRO CABANILLAS PRUDENCIO – ÁLVARO DE LAS HERAS FERNÁNDEZ
AMPLIACIÓN DE PROGRAMACIÓN AVANZADA
[Dirección de la compañía]



Introducción

La práctica que se nos presentó consistía en la realización de un programa que emulase el funcionamiento del popular juego 2048, basado en el movimiento de bloques numéricos que finaliza cuando el usuario obtiene el número 2048 en alguna de las casillas de la matriz.

El funcionamiento del programa a elaborar se llamaría 16384 y funcionaría de igual modo que el 2048, pero el número a obtener sería 16384. La herramienta para elaborar dicho programa sería CUDA, una plataforma de computación en paralelo que incluye un compilador y un conjunto de herramientas de desarrollo creadas por nVidia que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de nVidia.

De entre los requerimientos iniciales de la práctica destacan 4 fases: Realización de métodos referentes a la lógica del juego, realización del 16384 en memoria global, realización del 16384 por bloques y realización del 16384 por memoria compartida.

Así, lo primero que hicimos fue la realización de los métodos para implementar la lógica del juego en memoria global.

Realización de la lógica del juego en memoria global y un bloque

Posteriormente se detallarán todos los métodos de la práctica realizados en CPU, con su respectiva explicación de qué hacen y cómo lo hacen.

ImprimirMatrizVector: El objetivo de este método será imprimir el tablero del juego con sus respectivos elementos. Los parámetros de entrada que tendremos serán: puntero de entero al tablero, número de columnas del tablero y número de filas del tablero. Con dos for's anidados, uno para las filas y otro para columnas, recorreremos el tablero (anteriormente inicializado) y cambiaremos el color de las casillas en función del número que contengan con un switch. Tras cambiar el color, imprimiremos esa casilla y reiniciaremos el color para dejarlo a blanco y negro. Este proceso se seguirá de la misma forma para el resto de las casillas del tablero.

Análisis: Esta parte costó poco realizarla y se implementó rápido en el programa.

```
/**
 * Imprime un vector (que representa una matriz) como si fuera una matriz
 * @param p_matriz_vector Vector que se mostrara por pantalla
 * @param P_WIDTH_X Anchura de la matriz
 * @param P_WIDTH_Y Altura de la matriz
 */
void imprimirMatrizVector(int* p_matriz_vector, int p_num_columnas, int p_num_filas) {
    //Bucle para imprimir las filas y columnas
    for (int i = 0; i < p_num_filas; i++) {
        for (int j = 0; j < p_num_columnas; j++) {

            switch (p_matriz_vector[i * p_num_columnas + j]) { // Modifica el color
en el que se mostrarán los elementos
            case 0:
                Color(8, 8);
                break;
```

```

        case 2:
            Color(15, 0);
            break;
        case 4:
            Color(14, 0);
            break;
        case 8:
            Color(13, 0);
            break;
        case 16:
            Color(5, 15);
            break;
        case 32:
            Color(6, 0);
            break;
        case 64:
            Color(4, 0);
            break;
        case 128:
            Color(9, 0);
            break;
        case 256:
            Color(1, 0);
            break;
        case 512:
            Color(10, 0);
            break;
        case 1024:
            Color(2, 0);
            break;
        case 2048:
            Color(7, 0);
            break;
        case 4096:
            Color(8, 0);
            break;
        case 8192:
            Color(3, 0);
            break;
        case 16384:
            Color(15, 0);
            break;
        default:
            Color(0, 15);
            break;
    }
    printf("%d", p_matriz_vector[i * p_num_columnas + j]);
    printf("\t");
    //Color(0,15);

}
printf("\n");

}
Color(0, 15);
}

```

InicializarMatriz: El objetivo de este método será inicializar la matriz con múltiplos de dos en función de la dificultad del juego y las semillas generadas. Los parámetros que le entran a este método son: columnas y filas de la matriz, dificultad del juego y la cantidad de semillas en función de la dificultad (Semillas alta serán las semillas que entrarán si la dificultad es alta y semillas baja serán las semillas que entrarán si la dificultad es baja). Lo primero que hacemos en este método es definir una variable para el tablero y asignarle memoria con el malloc. Después almacenamos en una variable numero_semillas las semillas que se generarán y un vector con las posiciones de las semillas (posición_semillas), que tendrán un valor inicial. Utilizamos el srand para generar valores aleatorios en las semillas y evaluamos la dificultad que el usuario ha elegido. Si la dificultad es alta, el numero de semillas a generar (numero_semillas) será el número de semillas que se generan con la dificultad alta, que en el caso del 16384 son 8. Una vez establecido el número de semillas a generar, inicializamos el tablero con dos for's que utilizaremos para recorrer la matriz e ir rellenándola con ceros. Después pasamos a evaluar si el número de semillas es mayor que las casillas totales del tablero, si el número de semillas es mayor, el numero de semillas a generar será el tamaño total del tablero, es decir, rellenaremos todo el tablero sin dejar huecos. Por último, definiremos el contenido del vector que contendrá los índices generados aleatoriamente a través del método generarIndexSemilla, al que le pasaremos el número de filas, columnas y número de semillas a generar. Por último, una vez generados los índices dentro de la matriz, tendremos que colocar las semillas. El valor de las semillas vendrá dado por generarNumero, método que nos genera múltiplos de dos de forma aleatoria en función de la dificultad que se evaluará en esta memoria posteriormente. En el caso de la dificultad alta el valor de las semillas generadas inicialmente será de 2 o 4. De modo análogo haremos con la dificultad baja, solo cambiará el número de semillas a generar y el valor de éstas. Se generarán esta vez 15 semillas para facilitar al usuario las posibles combinaciones y los valores de éstas serán 2, 4 u 8. De esta forma, ya tendremos inicializado el tablero con las semillas y sus valores iniciales, es decir, tendremos el primer paso para jugar al 16384.

Análisis: Esta parte nos costó un poco llevarla a cabo porque teníamos dudas sobre cómo se generaban las semillas y en qué momento. Una vez resueltas estas dudas la tarea se llevó a cabo.

```
/**
 * Inicializa la matriz con multiplos de dos en funcion de la dificultad y semillas
 * @param p_num_columnas Anchura de la matriz
 * @param p_num_filas Altura de la matriz
 * @param p_dificultad indica el modo de dificultad elegido
 * @param p_semillas_alta numero de semillas que se crearan en nivel de juego alto
 * @param p_semillas_baja numero de semillas que se crearan en nivel de juego bajo
 */
int* inicializarMatriz(int p_num_columnas, int p_num_filas, bool p_dificultad, int
p_semillas_alta, int p_semillas_baja) {
    //Matriz que representa el tablero que se inicializa ademas se le asigna memoria
    int* tablero = (int *)malloc(p_num_columnas*p_num_filas * sizeof(int));
    //Numero de semillas que se generaran
    int numero_semillas = p_semillas_baja;
    //Vector con las posiciones de las semillas que contendran un valor inicial
    int* posicion_semillas;
    //Instante de tiempo que se usara para obtener numeros aleatorios
```

```

srand((unsigned int)time(NULL));
//Segun la dificultad generara unas semillas u otras
// Si la dificultad es alta
if (p_dificultad)
{
    //Semillas que tendra la dificultad alta
    numero_semillas = p_semillas_alta;
    //Inicializamos el tablero
    for (int i = 0; i < p_num_filas; i++)
    {
        for (int j = 0; j < p_num_columnas; j++)
        {
            tablero[i * p_num_columnas + j] = 0;
        }
    }
    //Si hubiera mas semillas que casillas se llena hasta el maximo
    if (numero_semillas > (p_num_columnas*p_num_filas)) {
        numero_semillas = p_num_columnas * p_num_filas;
    }
    //Vector con los indices generados aleatoriamente donde iran los valores
    posicion_semillas = generalIndexSemilla(p_num_columnas, p_num_filas,
numero_semillas);
    for (int i = 0; i < numero_semillas; i++)
    {
        //Se coloca en la posicion de la semilla el multiplo aleatorio de 2
        tablero[posicion_semillas[i]] = generarMultDosAle(p_dificultad);
    }
}
//Modo de dificultad baja
else {
    for (int i = 0; i < p_num_filas; i++)
    {
        for (int j = 0; j < p_num_columnas; j++)
        {
            tablero[i * p_num_columnas + j] = 0;
        }
    }
    //Si hubiera mas semillas que casillas se llena hasta el maximo
    if (numero_semillas > (p_num_columnas*p_num_filas)) {
        numero_semillas = p_num_columnas * p_num_filas;
    }
    //Vector con los indices generados aleatoriamente donde iran los valores
    posicion_semillas = generalIndexSemilla(p_num_columnas, p_num_filas,
numero_semillas);
    for (int i = 0; i < numero_semillas; i++)
    {
        //Se coloca en la posicion de la semilla el multiplo aleatorio de 2
        tablero[posicion_semillas[i]] = generarMultDosAle(p_dificultad);
    }
}
//Finalmente devolvemos el tablero
return tablero;
}

```

generalIndexSemilla: El objetivo de este método será generar un vector de enteros con los índices de las semillas para las casillas de la matriz. Los parámetros de entrada serán el número de filas y columnas del tablero y el tamaño del vector a generar. Lo primero que hacemos es definir un vector de enteros, que será el que devolveremos al final. Al definirlo le asignaremos memoria con el malloc. Tendremos que definir un primer índice para comenzar el proceso de asignación de índices, así que definiremos una variable index (que será un entero) cuyo valor será una casilla aleatoria de la matriz. Después tendremos un bucle while que irá generando el resto de los índices dentro de la matriz de forma aleatoria. Definiremos una variable contador que nos ayudará en el proceso de generación de índices, que controlará que no superemos el tamaño del vector. Lo primero que haremos en el bucle es controlar que el índice a generar no esté repetido con la función contiene, que evalúa si un determinado índice se encuentra ya en la matriz. Si no le contiene, generaremos el índice y daremos un nuevo valor aleatorio a index; repetiremos este proceso hasta generar todos los índices con valor único en la matriz. Finalmente devolveremos el vector de índices de la matriz, para posteriormente inicializarla.

Contiene: El objetivo de este método será comprobar si un valor se encuentra en un determinado vector. La usaremos en el método generarIndexSemilla para comprobar que el vector indexes no tiene valores repetidos. Los parámetros de entrada de este método serán un determinado valor que simbolizará el índice generado, un vector de enteros semillas que almacenará todos los índices ocupados hasta el momento y el tamaño de este vector. Tendremos un booleano estaContenido inicializado a false para partir inicialmente de que no está contenido el valor a evaluar. Posteriormente recorreremos el vector con un for y si el valor a evaluar coincide con alguno del vector pone estaContenido a true y devolvemos ese valor.

comprobarCasillasVacías: Este método será auxiliar de inicializar matriz y nos devolverá las casillas vacías que hay en el tablero, para posteriormente generar las semillas pertinentes. Esto lo hará recorriendo el tablero con dos for's y viendo si la casilla es 0 o no.

Análisis: Esta parte nos costó poco realizarla, ya que son métodos auxiliares de inicializarMatriz y se implementaron mientras se realizaba este.

modoManual: El objetivo de este método es emular el modo de juego en el que el usuario interactúa con el programa. Los parámetros que le entran serán el tablero, las filas y columnas de este, la dificultad elegida por el usuario, el número de semillas para dificultad baja y el número de semillas para dificultad alta; estos son los elementos que necesitaremos para jugar. Lo primero que haremos será definir una serie de variables que cumplan con las anteriores en el desarrollo del juego. Así, definimos:

- **Record:** llamará al método cargarRecord(), que comentaremos posteriormente en esta memoria. Este método devuelve un entero con la puntuación mas alta conseguida en el juego, almacenada en un archivo llamado record.
- **Vidas:** Será el número de vidas del usuario, según las especificaciones del enunciado de la práctica es 5.
- **Size:** Será el número total de hilos, el resultado de la multiplicación de filas por columnas.
- **Puntuación:** Será un entero en el que se irá almacenando la puntuación que vaya consiguiendo progresivamente el usuario.
- **Tecla:** Variable de tipo char que almacenará la tecla que pulse el usuario, para su interacción con el juego.

- Puntuación_anterior: Variable de tipo entero que utilizaremos para determinar el momento en el que el usuario pierde una vida y para el record.
- bloqueoEjeX y bloqueoEjeY: Variables bool que nos servirán de ayuda en el momento en el que el usuario pierde una vida.

Una vez definidas todas las variables que necesitamos, pasaremos a evaluar las acciones que se pueden llevar a cabo en la lógica del juego. Primero le daremos al usuario la opción de cargar una partida anterior, si el usuario responde que si llamaremos a cargarPartida() y cargaremos la partida que el usuario desee. Posteriormente, deberemos reservar memoria para las variables CUDA que irán al dispositivo: la puntuación y el tablero, ya que las necesitaremos en el desarrollo de los movimientos, que se realizarán en la GPU. Finalmente, tendremos el bucle del juego, que será un do while con la condición del while a true, ya que saldremos del bucle mediante breaks. Lo primero que hacemos es guardar la puntuación conseguida por el usuario como puntuación anterior y, posteriormente, mostramos el tablero. Mostramos también por pantalla tanto la puntuación como el record y las vidas.

```

Desea cargar partida?(Pulse y para cargar partida)

---.::: 16384 - THE GAME :::.---
=====
4      2      2      4      2
2      2      2      2      2
=====
Puntuacion: 0   Record: 4676
Vidas: <3 <3 <3 <3 <3

```

Después, evaluamos las vidas para comprobar si se puede seguir jugando. Si las vidas son menores que 0, le decimos al usuario que ha perdido la partida y liberamos memoria. Si se queda sin vidas y en el transcurso de la partida ha superado el record, le indicamos al usuario que ha superado el record y guardamos el nuevo record en el archivo record con el método guardarRecord(). Una vez hemos comprobado que puede seguir jugando, solicitamos al usuario un movimiento con el getch(). En esta solicitud deberemos comprobar con una condición si la tecla pulsada para el movimiento devuelve -32, ya que esto nos indica que es una flecha del teclado y requiere un tratamiento especial porque pilla dos pulsaciones de teclado en la misma tecla.

Ahora, en función de la tecla pulsada deberemos realizar una determinada acción, esto lo implementaremos con un switch que recibe como parámetro la tecla pulsada. Si se recibe una w o una flecha up, se pasa el tablero a la memoria de la gráfica, se realiza el cálculo en el kernel moverArriba, se devuelve el resultado a la CPU, obtenemos la puntuación una vez realizado el movimiento y rellenamos el tablero con su estado tras el movimiento realizado. Esto lo haremos así en el resto de los movimientos asd o flecha right, left o down. Cabe destacar que para las flechas tenemos que poner su correspondiente ASCII y por las letras basta con entrecomillarlas, ya que es un carácter.

```

case 'w':
    //Se pasa a memoria de la grafica el tablero
    cudaMemcpy(tablero_dev, p_tablero, size * sizeof(int),
cudaMemcpyHostToDevice);
    //Se realiza el calculo en el kernel
    moverArriba << <1, dimBlock >> > (tablero_dev, p_num_columnas,
p_num_filas, puntuacion_dev);
    //Se devuelve el resultado
    cudaMemcpy(p_tablero, tablero_dev, size * sizeof(int),
cudaMemcpyDeviceToHost);
    //Obtenemos la puntuacion despues del movimiento
    cudaMemcpy(&puntuacion, puntuacion_dev, sizeof(int),
cudaMemcpyDeviceToHost);
    //Despues de realizar el movimiento se rellena con valores
    p_tablero = rellenarTablero(p_tablero, p_num_columnas, p_num_filas,
p_dificultad, p_semillas_alta, p_semillas_baja, &lleno);
    break;

```

En el switch tendremos también dos condiciones para las teclas g y e. Si el usuario pulsa g guardaremos la partida justo en el instante en el que la ha guardado y si pulsa e se saldrá del juego. Antes de salir, se liberará memoria y se evaluará si el record ha sido superado; de ser así, se le indicará al usuario y se guardará el nuevo record.

Con esto tendremos terminado el switch que evalúa la tecla pulsada por el usuario, pero seguiremos en el bucle del juego. Lo siguiente a hacer dentro del bucle será evaluar si el usuario ha bloqueado el tablero (no se pueden realizar más movimientos). Para ello, tendremos una variable lleno de tipo bool que nos dirá si el tablero está lleno o no, ya que esta es una de las condiciones para que se bloquee el juego. Si está lleno, comprobaremos si entre movimientos se ha realizado algún cambio de puntuación, ya que si el usuario realiza los movimientos awsd y no se ha realizado ningún cambio de puntuación querrá decir que el tablero está bloqueado y no se pueden hacer más movimientos. De ser así, pondremos las variables bloqueoEjeX y bloqueoEjeY a true y el juego quedará bloqueado. Se le indicará al usuario que ha perdido y se le preguntará si quiere volver a jugar, en caso afirmativo se le restará una vida. Si no quiere jugar se le dará opción a guardar la partida por si quiere seguir jugando después y se evaluaría si ha superado el record antes de salir del juego.

Análisis: Con esto tendríamos toda la lógica del juego, parte de gran importancia en nuestro programa que, cabe destacar, nos costó realizar por errores que se daban en relación con el movimiento de los hilos. Para solucionar estos problemas tuvimos que dejar de lado el trabajo en solitario algunas tardes para trabajar en común en la universidad. Es el método que más tiempo nos llevó de la CPU.

modoAutomatico: El funcionamiento de este modo es similar al del modo manual en lo que se refiere a la lógica del juego, lo único que cambia es que los movimientos que se realizan no necesitan de ninguna tecla a presionar por el usuario, ya que los movimientos se realizan de forma aleatoria. En la pantalla se mostrará el movimiento adoptado aleatoriamente cada turno y se irá cambiando el tablero. El bloqueo del tablero y la gestión de vidas funciona exactamente igual que en el modo manual.



Análisis: Como realizamos el modo manual antes que el automático, lo único que tuvimos que hacer fue que los movimientos fuesen aleatorios, lo cual es muy fácil con un rand de 4. Si es 0 se moverá arriba, si es 1 a la izquierda, si es 2 abajo y si es 3 a la derecha.

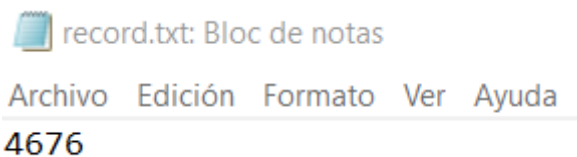
guardarPartida y cargarPartida: Gracias a estos métodos podremos implementar la parte que hace referencia al guardado y cargado de partidas dentro del juego. Para guardar la partida abriremos un archivo .txt, que se creará si no está creado cuando se ejecute el programa, y se guardarán los datos referentes a (en orden de aparición en el archivo) las vidas, puntuación, numero de columnas y filas de la matriz y el tablero entero en el estado en el que se encuentre. La partida guardada se



sobrescribirá si hay alguna partida guardada previamente. Para cargar la partida haremos un read del archivo .txt creado y recompondremos el tablero a partir de los datos del archivo.

Análisis: El problema que nos dieron estos métodos fue que no sabíamos cómo crear un archivo y leer y escribir en él. Solucionamos este problema a través de búsquedas en internet y, sobre todo, con el portal stack overflow. Una vez supimos leer y escribir el archivo, la realización de este apartado se llevó a cabo con éxito en poco tiempo.

guardarRecord y cargarRecord: Gracias a estos dos métodos podremos almacenar la puntuación más alta obtenida en el juego y cargarla al comienzo de cada partida. Lo haremos de forma similar a cargar y guardar partida, accederemos a un archivo que se creará con la ejecución del .cu si no está creado y haremos write sobre él para guardar el nuevo record, que se sobrescribirá si ya hay algún record guardado. Para cargar el record haremos read sobre el archivo record.txt y devolveremos el record que esté escrito en el archivo (si hay algún record escrito).



Análisis: La realización de estos métodos no nos llevó mucho tiempo hacerla, ya que teníamos hechos ya los de guardar y cargar partida y el funcionamiento es muy similar.

obtenerCaracteristicasCUDA: Este método nos servirá para obtener las características de la arquitectura CUDA del equipo en el que corramos el programa. Además, nos dirá si podemos ejecutar el programa o no, esto dependerá de si los hilos por bloque soportados por el equipo son menores que el tamaño de la matriz y de si se excede el tamaño máximo de memoria global. Aparte de las características que podemos obtener con nuestro método hemos implementado el porcentaje de aprovechamiento de los hilos que nos dirá cuantos de los hilos totales se van a usar en la ejecución del programa.

```
C:\Users\Alex\source\repos\16384\x64\Debug>16384.exe -m 2 5 5
¡Bienvenido al juego 16384!
Las características de su grafica son las siguientes:
Nombre: GeForce GTX 1050 - Capability Version: 6.1
Límites de hilos por bloque: 1024
Límites de hilos por SM: 2048
Límites de memoria global: 2147483648
Límites de memoria compartida: 49152
Límites de registros: 65536
Número de multiprocesadores: 5
Las características de la matriz a emplear:
Cantidad de hilos a emplear: 25
Cantidad de memoria que se emplea: 104
Porcentaje de aprovechamiento de hilos: 2.44 %
```

Análisis: Este método no nos llevó mucho tiempo, ya que ambos miembros del grupo hicimos en su día el ejercicio de obtener las características de la tarjeta y únicamente fue comparar los programas de ambos integrantes y seleccionar los datos que mejor se ajustaban al problema. Las condiciones que evalúan si se puede correr el programa en un equipo o no las sacamos de la teoría de la asignatura.

Color: Este método lo único que hace es cambiar de color la consola para un determinado texto. Lo utilizamos con los valores del tablero y para elementos gráficos del programa.

Análisis: Este método no nos llevó mucho tiempo, únicamente tuvimos que buscar los colores y el como cambiarlos en la consola en internet e implementarlo en nuestro programa.

Una vez se han explicado todos los métodos que se realizan en la CPU, vamos a proceder a explicar los que se realizan en GPU de NVIDIA. Estos métodos son los referentes a los movimientos en el tablero.

moverAbajo, moverArriba, moverIzquierda, moverDerecha: Esta función se encargará de mover los valores del tablero y sumarlos en el caso de que sean iguales. Dividimos el método en 3 partes: primero quitamos los 0's de la matriz, para ello cada hilo irá desplazando donde corresponda los valores si no son 0 en los que sean 0; después, una vez tenemos juntos los valores, evaluamos si son iguales y si lo son, los sumamos y dejamos la casilla anterior a 0 y, finalmente, se vuelven a quitar los ceros tras sumar para dejar un resultado más limpio. Para obtener la puntuación nos aseguramos de cada hilo sume en orden la suma de puntos a la variable puntuación que le pasamos como parámetro.

Análisis: Esta es la parte de la práctica que más nos costó, tuvimos problemas con los hilos, ya que a veces varios accedían a varias filas, por ejemplo. Para suplir este problema tuvimos que poner numerosas condiciones que controlasen los accesos de los hilos y el cómo se mueven por el tablero.

Main: Como nuestro programa se ejecuta a través de la consola de Windows, tuvimos que realizar cambios en el main para que esto fuese posible. Esto lo conseguimos con los parámetros argc y el argv, con ellos evaluamos el modo de juego, la dificultad y el número de filas y columnas que el usuario desea. En función de estos parámetros lanzamos el juego.

Realización de la lógica del juego en memoria global y múltiples bloques

La adaptación del juego a memoria por bloques ha sido prácticamente directa. Lo primero que hicimos fue añadir en los movimientos las variables fila y columna que utilizaremos para calcular la posición del hilo dentro de la matriz a partir del ID del bloque, del hilo y del TILE_WIDTH (tal y como se especifica en la teoría de la asignatura). A partir de estas variables, hemos introducido una condición para controlar que no trabajamos con hilos fuera del rango de nuestra matriz. Así, tendremos bien definidos los límites dentro del bloque, de esta forma evitaremos acceder a posiciones nulas del bloque.

```
// Calculamos la fila en la matriz correspondiente al hilo y su bloque
int fila = blockIdx.y * P_TILE_WIDTH + threadIdx.y;
// Calculamos la columna en la matriz correspondiente al hilo y su bloque
int col = blockIdx.x * P_TILE_WIDTH + threadIdx.x;
// Posición del hilo del bloque en la matriz
int posicion = fila + col * p_num_columnas;
```

Realización de la lógica del juego en memoria compartida y múltiples bloques

Esta parte no ha podido ser realizada, aunque si se ha intentado. El planteamiento que hemos seguido para poder emplear memoria dinámica ha sido el de comunicar los hilos límites en las teselas mediante memoria global. Hay que realizar esto porque la memoria compartida se limita al bloque, no pudiendo compartir entre bloques del mismo grid. Para intentar solucionar esto se recurre a memoria global para que en los límites de las teselas se acceda a global permitiendo comunicarse a coste de un mayor tiempo de acceso. Esto se ha realizado, pero no se han obtenido datos coherentes al duplicar casillas y eliminar valores al realizar el movimiento.

En definitiva, hubiera consistido en comunicar los límites de las teselas mediante el uso de global, porque la memoria compartida se limita al bloque.