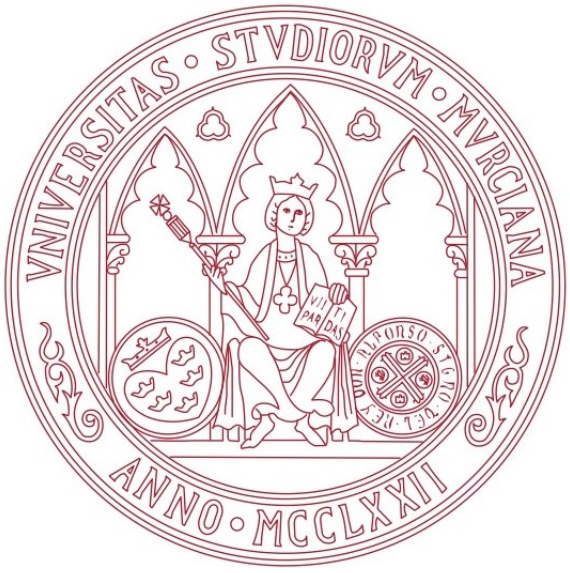

Programación de Arquitecturas Multinúcleo

PROGRAMACIÓN BÁSICA DE UNA GPU

Francisco Arnaldo Boix Martínez - 52041159W

Marzo 2023

Índice

1. Programa `vectorAdd`

1.1. Cálculo del número de bloques necesarios

```
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
```

Se usa un pequeño *truco* matemático para irse al entero superior. Es decir, asegurarse de que siemore hay bloques suficientes. AUnque haya alguno que no se haya llenado

1.2. ¿Qué sucede si el número total de elementos de los vectores no es múltiplo del tamaño del bloque de hilos?

El programa sigue siendo correcto, pero vemos el efecto previamente comentado, se coge un bloque más, aunque este únicamente contenga 1 hilo efectivo.

1.3. Ampliación del programa: `D = A - B`

El fichero `vectorSubstract.cu` adjuntado muestra el resultado. Para recapitular, los pasos seguidos han sido:

1. Reservar memoria para un nuevo array destino en la pila del proceso (CPU).

```
// Allocate the host output vector D
float *h_D = (float *)malloc(size);

```

2. Comprobar que el valor devuelto por malloc, es correcto

```
    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL || h_D == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }
```

3. Reservar memoria en la memoria global de la GPU.

```
    // Allocate the device output vector D
    float *d_D = NULL;
    err = cudaMalloc((void **)&d_D, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector D (error
   code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

4. Crear el nuevo kernel

```
__global__ void
vectorSubstract(const float *A, const float *B, float *C, int
    numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    int bdx = blockDim.x;
    int bix = blockIdx.x;
    int tix = threadIdx.x;

    if (i < numElements)
    {
        //  printf("threadIdx.x=%d, i=%d\n",threadIdx.x,i);
        C[i] = A[i] - B[i];
    }
}
```

5. Llamar al nuevo kernel

```
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
    numElements);
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorAdd kernel (error
    code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

6. Llamar al nuevo kernel con el nuevo vector destino

```
    vectorSubstract<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_D
    , numElements);
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorAdd kernel (error
    code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

7. Copiar el nuevo vector resultado del dispositivo a la CPU.

```
    // Copy the device result vector in device memory to the host
    result vector
    // in host memory.
    printf("Copy output data from the CUDA device to the host memory\
    n");
    err = cudaMemcpy(h_D, d_D, size, cudaMemcpyDeviceToHost);

```

```
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector C from device to host
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

8. Cambiar la condición para las pruebas.

```
if (fabs(h_A[i] - h_B[i] - h_D[i]) > 1e-5)
```

9. Liberar memoria tanto en el host como el dispositivo.

```
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector D (error code %
    s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

...

    free(h_D);
```

2. Programa `cuda_template`

2.1. Explicación del kernel `foo`

Como resultado de ejecutar el kernel obtenemos un vector de tamaño `num_hilos` en cuyas posiciones tenemos su `thread_id` dentro de la malla sumado a una constante.

El kernel hace un par de cálculos innecesarios, como por ejemplo:

```
shared_mem[tidb] = gid_d[tidg];
```

2.2. Analiza si el tamaño de la memoria compartida es correcto.

El tamaño de la memoria compartida es de tamaño `shared_mem_size`, el cual se calcula del siguiente modo:

```
shared_mem_size = block.x * block.y * sizeof(int);
```

// calla nene q esta correcto

Sin embargo, esto es incorrecto puesto que se accede a la memoria compartida como si tuviese `nPos` elementos, o `nBytes` bytes. Esto es así porque este es el número máximo de hilos que dispondrá el programa.

```
nPos = tam_grid_x * tam_grid_y * tam_block_x * tam_block_y;
    nBytes = nPos * sizeof(int);
```

2.3. Podrían eliminarse las llamadas a `__syncthreads()`

Ambas llamadas se podrían eliminar, porque el código no requiere un punto de sincronización de los hilos donde se llaman a la función. Esto es así porque los cálculos que se están llevando a cabo son totalmente **independientes**.

2.4. Modificación bloques de hilos tridimensionales

Lo que se pretende con esta modificación es alcanzar una organización de los hilos similar a las de la siguiente Figura, donde cada uno de los cubitos dentro de cada cubo grande (Bloque) representa un **hilo**.

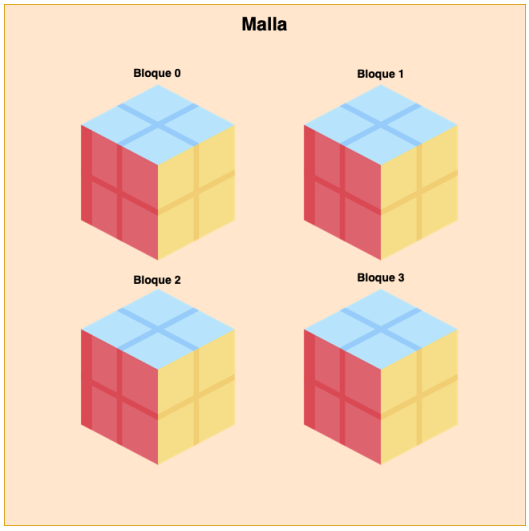

Figura 1: Disposición lógica tridimensional de los hilos dentro de cuda

En la versión por defecto, en vez de tener bloques que sean **cubos**, teníamos bloques que eran **cuadrados** (dos dimensiones). El resultado final se puede observar en el archivo adjuntado `cuda_template_3d.cu`.

Para añadir una tercera dimensión dentro de cada uno de los bloques, deberemos tener numeroso cuidado a la hora de calcular el **id** de cada uno de los **threads** a la hora de operar con el vector.

Para ello, añadimos la tercera dimensión a los cálculos. Notese que no hay que tocar el cálculo de `tidg` (Thread id Grid) puesto que el cálculo de los bloques dentro del grid es el mismo, ya que no ha cambiado la dimensionalidad.

```
int blockSize = blockDim.x * blockDim.y * blockDim.z;

// global thread ID in thread block
int tidb = (threadIdx.y * blockDim.x * blockDim.z + threadIdx.z * blockDim
    .x + threadIdx.x );
```

Sumado a esto, tendremos que:

1. Pedir el parámetro por la línea de comandos en caso de que así lo estimemos oportuno haciendo uso de la operación `getCmdLineArgumentInt()`.
2. Tener en cuenta la nueva dimensión para el cálculo del número de elementos que tendrá el vector.

```
int nPos = tam_grid_x * tam_grid_y * tam_block_x * tam_block_y *
   tam_block_z;

```

3. Cambiar las dimensiones de `block` para hacer la llamada al kernel.

```
dim3 block(tam_block_x, tam_block_y, tam_block_z);

```

4. Reflejar la tridimensionalidad en el cálculo de la memoria compartida.

```
shared_mem_size = block.x * block.y * block.y * sizeof(int);

```

5. Cambiar los `printf()` para poder ver la información correcta y actualizada.

3. Programa `cuda_vectorReduce`

Para poder entender cada uno de las secciones acerca de este programa, es importante entender qué problema resuelve y cómo lo lleva acabo.

El problema a resolver es la suma de todos los elementos de un vector, en vez de hacerlo secuencialmente con un recorrido, lo haremos dentro de una GPU aprovechando al máximo la arquitectura que disponemos por debajo.

Para sumar el vector entero, se va a ir haciendo por bloques. de hilos. Si observamos la Figura 2, cada bloque de hilos se encargará de un color. Como vemos, el algoritmo tiene en cuenta el tamaño de **bloque** y va reduciendo su tamaño lógico a la mitad, cada vez que va haciendo las sumas parciales de los elementos correspondientes. Este proceso iterativo se repetirá hasta que se llegue a un bloque en el cual tengamos todas las sumas parciales de todos los elementos de un mismo bloque, lo que equivaldría al último bloque de cada color. En este punto bastará con sumar el primer bloque de cada color y obtendremos la suma final.

Esto se lleva a cabo teniendo en cuenta el tamaño de bloque, el cual se va partiendo por la mitad, y computando los elementos uno a uno de la primera mitad, frente a la segunda mitad. En el caso que el número de elementos no sean pares también se tiene en cuenta, siempre y cuando el tamaño de bloque sea **potencia de dos**.

Para darnos cuenta, prestemos atención a los bloques de color azul. Podemos observar que únicamente hay 3 elementos a sumar dentro del bloque. Esto podría parecer una situación

problemática, sin embargo, el segundo elemento dentro del bloque azul se sumará con el valor constante **cero**. Es importante ver el detalle de que estos dos elementos (`bloque_azul[1] + bloque_azul[3]`) o (`vector[12] + vector[15]`) se sumarán porque el tamaño de bloque es **4**. Por tanto, podemos concluir que aunque el tamaño del vector sea impar el algoritmo funciona.

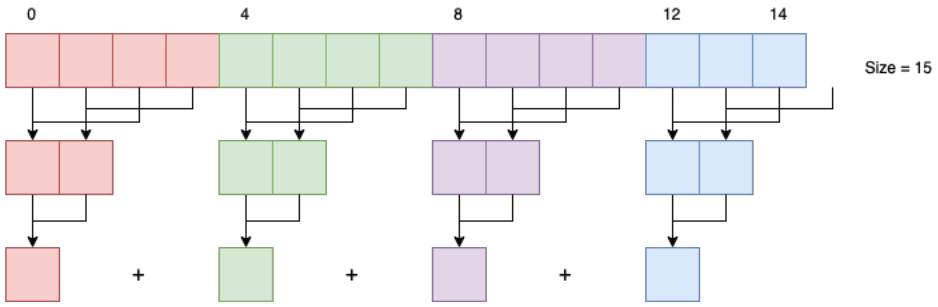

Figura 2: Funcionamiento del algoritmo.

3.1. Cálculo del número de bloques y de hilos por bloque necesarios.

```
1    // process command line arguments
2    n=getCmdLineArgumentInt(argc, (const char **) argv, (const char *) "n"
    )?:n;
3    bsx=getCmdLineArgumentInt(argc, (const char **) argv, (const char *) "
    bsx")?:bsx;
4
5    nBytes = n * sizeof(float);
6
7    dim3 grid( (n%bsx) ? (n/bsx)+1 : (n/bsx) );
8    dim3 block(bsx);
```

- **Número de bloques:** Coge el entero superior a la división del tamaño del vector entre el número de hilos por bloque. Es decir, hace un reparto equitativo.
- **Número de hilos por bloque:** Lo deja a elección del usuario.

3.2. ¿Qué sucede si el número total de elementos del vector no es múltiplo del tamaño del bloque de hilos?

Como comentaba en la Sección 3, si el número total de elementos del vector no es múltiplo del tamaño del bloque de hilos, se cogerá un bloque que en que habrá posiciones sin ocupar, y ya hemos visto como el algoritmo sigue teniendo en cuenta ese caso.

3.3. ¿Qué problema ocurre si el tamaño del bloque de hilos no es potencia de 2? Modifica el código para solucionar este problema.

Lo que ocurre es que el programa deja de ser correcto. Vemos que el programa no supera los test ya que no termina de pasar todos los asertos.

Entendamos el trasfondo del problema: Si hacemos un poco de **zoom** sobre el vector y nos centramos en el cálculo de un solo bloque, observemos la Figura 3 en la cual se ilustra la ejecución del algoritmo para un tamaño de bloque de **10**.

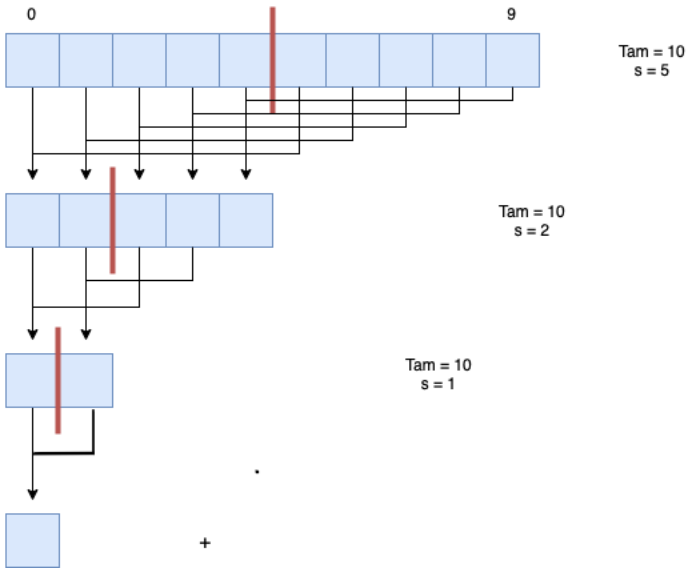

Figura 3: Funcionamiento del algoritmo.

Como podemos observar, cuando la variable `s` tiene un valor **impar**, al hacer la división del **subvector** por la mitad, no todos los elementos tienen mapeados su correspondiente en la otra mitad. Esto ocurre porque cuando tenemos un tamaño de bloque impar, el punto para partir el vector por la mitad se dispone una posición a la izquierda en comparación a lo que debería ocurrir para simular el caso que comentábamos en la Sección 3.

Por este motivo, es necesario añadir esta comprobación dentro del bucle principal del algoritmo:

```
1    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
2    {
3        if (tidb < s) {
4            sdata[tidb] += sdata[tidb + s];
5        }
6        // Modificacion
7        if (s % 2 != 0 && s > 1 && tidb == 0) {
8            atomicAdd(&sdata[0], sdata[s - 1]);
9        }
10       __syncthreads();
11   }
```

Es importante recalcar que esta operación únicamente la llevará acabo un hilo, puesto que solamente es necesario que se sume una vez.

Si solamente añadimos esta modificación nos funcionaría para tamaños de bloque que son **pares**, pero **no** son **potencias de 2**, puesto que en alguna iteración acabarán en un número impar, por la propia definición de la factorización de los números.

Esto es así por como está definido el bucle principal del algoritmo, el cual se inicializa la variable de control `s` a `blockDim.x / 2`. Por tanto, si el tamaño de bloque introducido por el usuario es directamente impar, tendremos que hacer esta mismo modificación antes de empezar con el bucle principal del algoritmo. Podemos verlo de ilustrado en la Figura 4

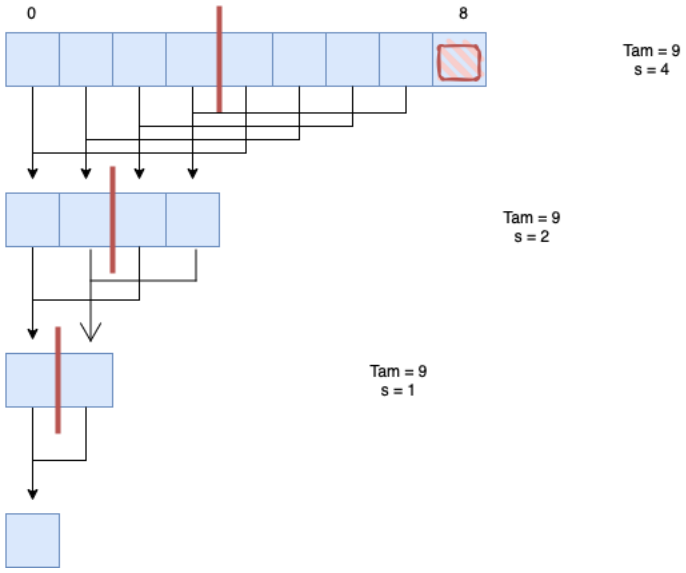

Figura 4: Funcionamiento del algoritmo.

Por tanto, añadiremos esta comprobación antes de este bucle.

```
if (blockDim.x % 2 != 0 && blockDim.x > 1 && tidb == 0)
{
    atomicAdd(&sdata[0], sdata[blockDim.x - 1]);
}
```

3.4. Analiza el código del kernel y explica el uso que hace de la memoria compartida.

Como sabemos, la memoria **compartida** es la memoria **lógica** que comparten **todos** los hilos dentro de un **bloque**. Esta memoria tiene como característica el ser mucho mas rápida que la **local** y la **global**.

En concreto, dentro del kernel cada hilo lleva el dato que le toca sumar desde la memoria global a la memoria compartida. Como esa memoria compartida únicamente tiene ámbito dentro de un bloque de hilos, únicamente hay que llevar cuidado con los accesos a esas variables dentro del mismo bloque de hilos.

3.5. ¿Podría eliminarse la primera llamada a `__syncthreads` en el código del kernel? ¿Y la segunda?

Ninguna de las llamadas a `__syncthreads` puede eliminarse, porque los cálculos que se están realizando dependen del resultado del de otros. Únicamente podríamos eliminarlas si nos aseguramos que el número de hilos que se vana a ejecutar dentro del kernel no supera el tamaño de **warp**. Esto es así puesto que sabemos que todos se van a ejecutar a la misma vez y no vamos a tener esos problemas de sincronización.

3.6. En este ejemplo, el kernel no completa la reducción en la GPU. Modifica el código para que la reducción se complete, de manera eficiente, en la GPU, evitando así la intervención de la CPU. Para ello, utiliza una reducción atómica final.

Para poder hacer todo el cómputo dentro de la GPU y abstraer a la CPU de cualquier tipo de cálculo, bastará con eliminar el bucle `for` final que hace la CPU, y añadir un código extra dentro del kernel `vectorReduce`. Para que el resultado sea correcto, el primer hilo de cada bloque suma la suma de todos los elementos dentro de su bloque, a la cantidad **global**. Nótese que ya es no es necesario hacer uso de un bucle `for` puesto que cada uno de los hilos con id 0, de cada bloque sumará la parte que le toca, haciendo uso de una instrucción **atómica** para garantizar el acceso en **exclusión mutua**.

```
1    // each block's first thread will add it's
2    if (tidb == 0)
3    {
4        atomicAdd(reduce_d, sdata[0]);
5    }
```

3.7. Modifica el código original para que la reducción se complete, de manera eficiente, en la GPU, evitando así la intervención de la CPU. En este caso, sin utilizar una reducción atómica final.

Del mismo modo que anterior, haremos lo mismo, pero sin hacer uso de la instrucción `atomicAdd`. Ahora, como no podemos hacer uso de una instrucción **atómica**, no podemos seguir la misma estrategia que antes, en vez de asignar la tarea de sumar a cada uno de los hilos con id 0 de cada uno de los bloques, la responsabilidad es asignada a un **único** hilo dentro de toda la malla de bloques. Este hilo se encargará de hacer un cálculo muy similar al que realizaba antes la **CPU**.

```
1    // write result for this block to global memory
2    if (tidb == 0)
3    {
4        reduce_d[blockIdx.x] = sdata[0];
```

```
    }

    if (tidg == 0)
    {
        // compute final stage
        for (int i = 1; i < gridDim.x; i++)
            reduce_d[0] += reduce_d[i];
    }
```

3.8. Estudio experimental de las últimas dos versiones.

Fijémonos en el Cuadro 1 y veamos los tiempos relativos al uso de una instrucción atómica para realizar la reducción final en la GPU. Esta versión corresponde a la Sección 3.6.

Vector Length	Block Size	Time
512	1	0.050016
	2	0.030080
	4	0.028416
	8	0.027968
	16	0.027648
1024	1	0.043328
	2	0.051296
	4	0.046048
	8	0.028160
	16	0.030816
2048	1	0.074528
	2	0.052800
	4	0.041504
	8	0.033024
	16	0.029248
4096	1	0.093216
	2	0.060352
	4	0.043136
	8	0.034304
	16	0.033472

Cuadro 1: Tiempos utilizar una instrucción atómica

Veamos ahora la tabla de tiempos relativa a la versión que **no** usa una instrucción **atómica**, correspondiente a la Sección 3.7. Véase el Cuadro 2.

4. Programa `escalar`

El producto escalar de dos vectores se define como: $\vec{a} \cdot \vec{b} = a_1 * b_1 + ... + a_n * b_n$

Vector Length	Block Size	Time
512	1	0.076576
	2	0.064800
	4	0.049824
	8	0.045088
	16	0.034624
1024	1	0.121792
	2	0.075328
	4	0.054176
	8	0.043936
	16	0.037696
2048	1	0.206176
	2	0.119744
	4	0.091264
	8	0.055360
	16	0.043584
4096	1	0.380576
	2	0.222528
	4	0.119904
	8	0.075872
	16	0.055328

Cuadro 2: Tiempos sin utilizar una instrucción atómica

Esta operación matemática es la que se pretende implementar en este programa llamado `escalar`. Para ello, reutilizaré el kernel `vectorReduce` corregido de la Sección 3.

Para implementarlo, he seguido los siguientes pasos:

1. He declarado las variables necesarias dentro tanto para el host, como la GPU.

```
    float *vector_h, *wector_h, *scalar_h, *reduce_h; // host data
    float *vector_d, *wector_d, *scalar_d, *reduce_d; // device data

```

2. Las he inicializado correctamente:

```
    // allocate host memory
    vector_h = (float *) malloc(nBytes);
    wector_h = (float *) malloc(nBytes);
    scalar_h = (float *) malloc(nBytes);

    for(int i = 0; i < n; i++)
    {
        vector_h[i] = (float) 1.0;
        wector_h[i] = (float) 2.0;
    }
    reduce_h = (float *) malloc(sizeof(float));
    bzero(reduce_h, 1 * sizeof(float));
```

```
13
14      // allocate device memory
15      checkCudaErrors(cudaMalloc((void **) &vector_d, nBytes));
16      checkCudaErrors(cudaMalloc((void **) &wector_d, nBytes));
17      checkCudaErrors(cudaMalloc((void **) &scalar_d, nBytes));
18      checkCudaErrors(cudaMalloc((void **) &reduce_d, sizeof(float)));
19
20      // copy data from host memory to device memory
21      checkCudaErrors(cudaMemcpy(vector_d, vector_h, nBytes,
        cudaMemcpyHostToDevice));
22      checkCudaErrors(cudaMemcpy(wector_d, wector_h, nBytes,
        cudaMemcpyHostToDevice));
23      checkCudaErrors(cudaMemset(reduce_d, 0, sizeof(float)));
```

3. He declarado una función auxiliar dentro del cómputo que realiza la cpu apoyándome en la directiva `__device__`.

```
1  __device__ void vectorScalarProduct(const float *vector_d, const
       float *wector_d, float *scalar_d, int n)
2  {
3      unsigned int tidg = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if (tidg < n)
6      {
7          scalar_d[tidg] = vector_d[tidg] * wector_d[tidg];
8      }
9
10 }
11
```

Esta función se usa dentro del kernel del ejercicio anterior, como paso previo a la inicialización de la memoria compartida dentro del kernel.

```
1  // load shared memory
2      vectorScalarProduct(vector_d, wector_d, scalar_d, n * blockDim.x)
       ;
3
4
5      sdata[tidb] = (tidg < n) ? scalar_d[tidg] : 0;
```

4. Se comprueba que el resultado sea correcto:

```
1
2      // check result
3      assert(*reduce_h == (float) 2 * n);
4
```

5. Si el resultado es correcto y no ocurre ningún fallo a la hora de liberar memoria el programa termina de manera correcta.

```
1      // free memory
2      free(vector_h);
3      free(wector_h);
4      free(scalar_h);
```

```
    free(reduce_h);
    checkCudaErrors(cudaFree((void *) vector_d));
    checkCudaErrors(cudaFree((void *) wector_d));
    checkCudaErrors(cudaFree((void *) scalar_d));
    checkCudaErrors(cudaFree((void *) reduce_d));

    printf("\nTest PASSED\n");

```

4.1. Tiempos dentro del programa

El programa debe calcular y mostrar el tiempo de ejecución total, así como el tiempo desglosado en tiempo de comunicación host-device, tiempo de comunicación device-host y tiempo de cálculo en GPU. Para ello, bastará con agregar las siguientes líneas antes y después de la región de nuestro programa que queramos medir. Para ello, veamos como se mide el tiempo de comunicación del **host** con el **dispositivo**:

```
    //create htod events
    checkCudaErrors(cudaEventCreate(&start_htod,0));
    checkCudaErrors(cudaEventCreate(&stop_htod,0));


    checkCudaErrors(cudaEventRecord(start_htod,0));

    // copy data from host memory to device memory
    checkCudaErrors(cudaMemcpy(vector_d, vector_h, nBytes,
   cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(wector_d, wector_h, nBytes,
   cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemset(reduce_d, 0, sizeof(float)));


    checkCudaErrors(cudaEventRecord(stop_htod, 0));
    cudaEventSynchronize(stop_htod);   // block until the event is
   actually recorded
    checkCudaErrors(cudaEventElapsedTime(&htod_time, start_htod, stop_htod
   ));

    printf("Host to device time: %lf (ms)\n\n", htod_time);
```

5. Repositorio GitHub

Todo el código mostrado a lo largo de este documento puede ser consultado pinchando aquí. Voy a pasar a comentar los contenidos de las distintas ramas, donde se han ido desarrollando cada una de las versiones.

- **master:** Contiene el código *virgen*, la versión correcta del algoritmo de `cuda_vectorReduce` y el ejercicio `escalar`, correspondiente a la Sección 4
- **template-tridimensional:** corresponde con la modificación del código de `cuda_template` disponible en la Sección 2.4.
- **vectorAdd-resta:** disponible en la Sección 1.3.
- **vectorReduce-apartadog:** correspondiente a la versión sin atomizar de `cuda_template` disponible en la Sección 3.7
- **vectorReduce-odd:** Correción del algoritmo de `cuda_vectorReduce` que admite tamaños de bloque que no sean potencia de dos.
- **vectorReduce-atomic:** Parte de la versión correcta del algoritmo y corresponde a la Sección 3.6.
- **tiempo:** parte de la última versión de la rama `master` y calcula el tiempo de copia entre el host y el dispositivo y viceversa.