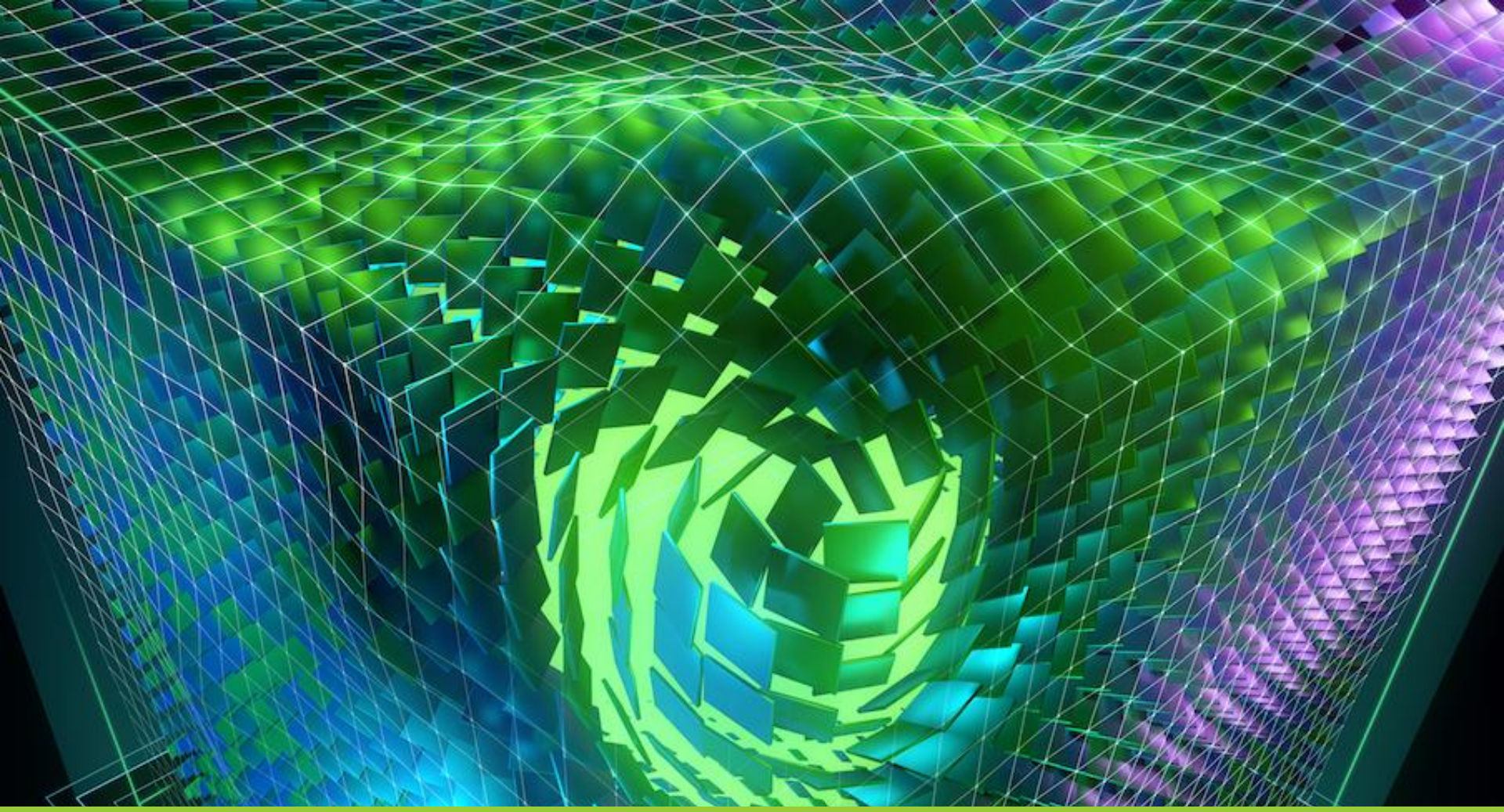


INF351 – Computación de Alto Desempeño

Memoria Constante y Texturas

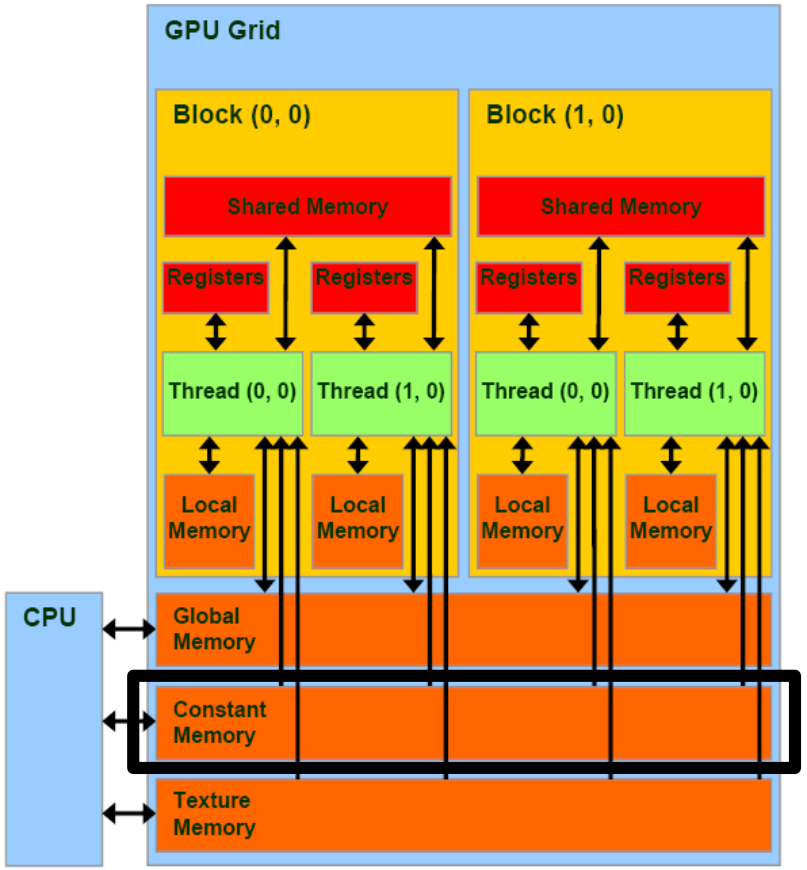
PROF. ÁLVARO SALINAS

Memoria Constante

Uno de los tipos de memoria del dispositivo.

Físicamente, se encuentra en el espacio de VRAM.

Solamente puede ser leída desde un kernel y, al igual que la memoria global, es visible para todas las hebras en ejecución.



Memoria Costante

La memoria constante es un tipo de memoria de la GPU ubicada en el espacio de VRAM junto a la memoria global. Hay 64 KB disponibles para memoria constante en un dispositivo.

Las hebras solo pueden leer desde memoria constante, pero no escribir en ella.

Dada su ubicación, la latencia de un acceso a memoria constante es similar a la de un acceso a memoria global. La diferencia radica en que la memoria constante es llevada a un constant cache, cuya latencia de lectura es casi tan rápida como el uso de registros.

Existen casos especiales en donde su uso conlleva mejoras de desempeño.

¿Cómo funciona?

Cuando una hebra intenta a acceder a un dato en memoria constante, éste es leído desde la VRAM y llevado al constant cache mencionado anteriormente.

Esto significa, que la primera lectura de un dato presenta una latencia alta, similar a la de memoria constante.

Una vez en caché, los datos pueden ser accedidos con menor latencia.

Finalmente, la cantidad de transacciones de memoria que necesitan las 32 hebras de un warp para leer datos desde el caché, es igual a la cantidad de direcciones de memoria distintas accedidas.

¿Cuándo se usa?

Mientras más hebras deseen leer el mismo dato, mejor será el desempeño de la memoria constante en comparación a la memoria global.

El caso ideal es cuando todas las hebras necesitan leer el mismo dato en su ejecución, como por ejemplo, cuando tenemos valores constantes a lo largo de la aplicación que no pueden ser determinados en tiempo de compilación.

La memoria constante no está relacionada a la coalescencia de los accesos, por lo que si cada hebra leerá un único dato, entonces usar accesos coalescentes a memoria global serán nuestra mejor opción.

Manejo de Constant Memory

Para crear variables residentes en memoria costante debemos utilizar la etiqueta *__constant__*.

La memoria constante solo puede ser asignada de forma estática. Su asignación debe estar en global scope. Un ejemplo es:

__constant__ int a[1000];

con lo que creamos un arreglo de 1000 enteros que serán visibles por todas las hebras de la aplicación.

Manejo de Constant Memory

De forma similar al uso de *cudaMemcpy()*, existe una función para copiar datos desde y hacia memoria constante.

Para copiar hacia memoria constante utilizaremos:

cudaMemcpyToSymbol(const_ptr, src, size, offset, cudaMemcpyKind);

Mientras que para copiar desde memoria constante tenemos:

cudaMemcpyFromSymbol(dst, const_ptr, size, offset, cudaMemcpyKind);

aunque no lo usaremos mucho.

Ejemplo

```
__global__ void kernel(int *src, int *dst, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N) dst[tId] = src[tId];
}
int main(){
    int N = 10000, block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);
    int *dst; cudaMalloc(&dst, N * sizeof(int));
    int *src; cudaMalloc(&src, N * sizeof(int));
    int *host_src = new int[N];
    fill_array(host_src); // fill the input data
    cudaMemcpy(src, host_src, N*sizeof(int), cudaMemcpyHostToDevice);
    kernel<<<grid_size, block_size>>>(src, dst, N);
    delete[] host_src;
    cudaFree(src);
    cudaFree(dst);
    return 0;
}
```

Ejemplo

```
__global__ void kernel(int *src, int *dst, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N) dst[tId] = src[tId];
}
int main(){
    int N = 10000, block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);
    int *dst; cudaMalloc(&dst, N * sizeof(int));
    int *src; cudaMalloc(&src, N * sizeof(int));
    int *host_src = new int[N];
    fill_array(host_src); // fill the input data
    cudaMemcpy(src, host_src, N*sizeof(int), cudaMemcpyHostToDevice);
    kernel<<<grid_size, block_size>>>(src, dst, N);
    delete[] host_src;
    cudaFree(src);
    cudaFree(dst);
    return 0;
}
```

Ejemplo

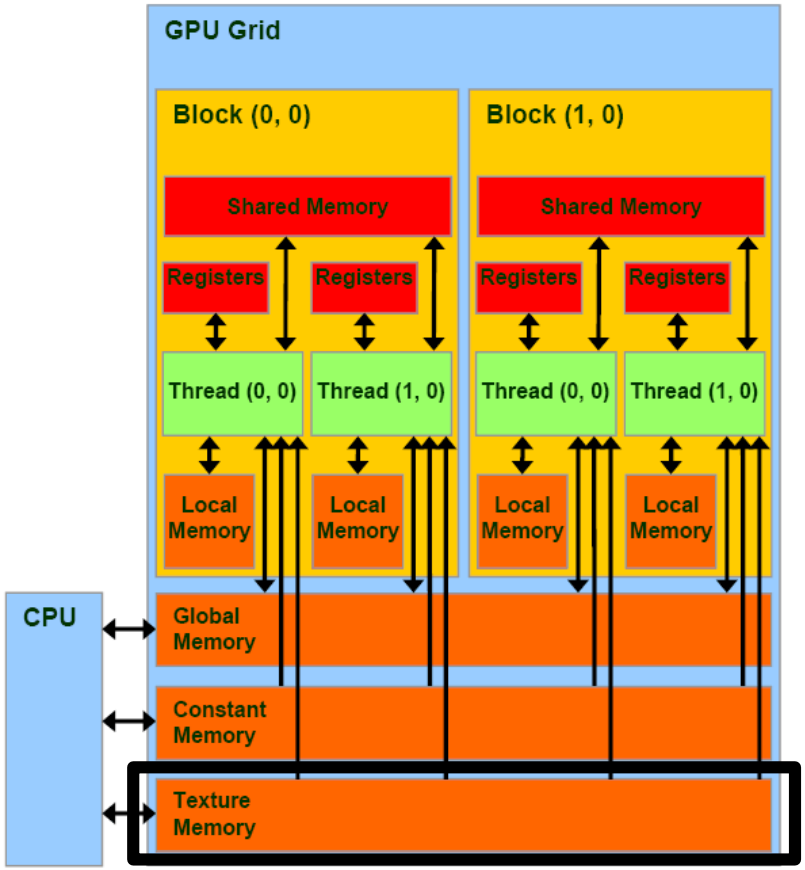
```
__constant__ int src[10000];
__global__ void kernel(int *dst, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N) dst[tId] = src[tId];
}
int main(){
    int N = 10000, block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);
    int *dst; cudaMalloc(&dst, N * sizeof(int));
    int *host_src = new int[N];
    fill_array(host_src); // fill the input data
    cudaMemcpyToSymbol(src, host_src, N*sizeof(int), 0, cudaMemcpyHostToDevice);
    kernel<<<grid_size, block_size>>>(dst, N);
    delete[] host_src;
    cudaFree(dst);
    return 0;
}
```

Texturas

Otro de los tipos de memoria del dispositivo.

Al igual que la memoria global y la memoria constante, se encuentra en el espacio de VRAM.

Solamente puede ser leída desde un kernel y es visible para todas las hebras en ejecución.



Texturas

Las texturas son un tipo de memoria de la GPU ubicada en el espacio de VRAM junto a la memoria global y la memoria constante. En realidad, las texturas corresponden a espacios de memoria global declarados como texturas.

Las hebras solo pueden leer desde una textura, pero no escribir en ella.

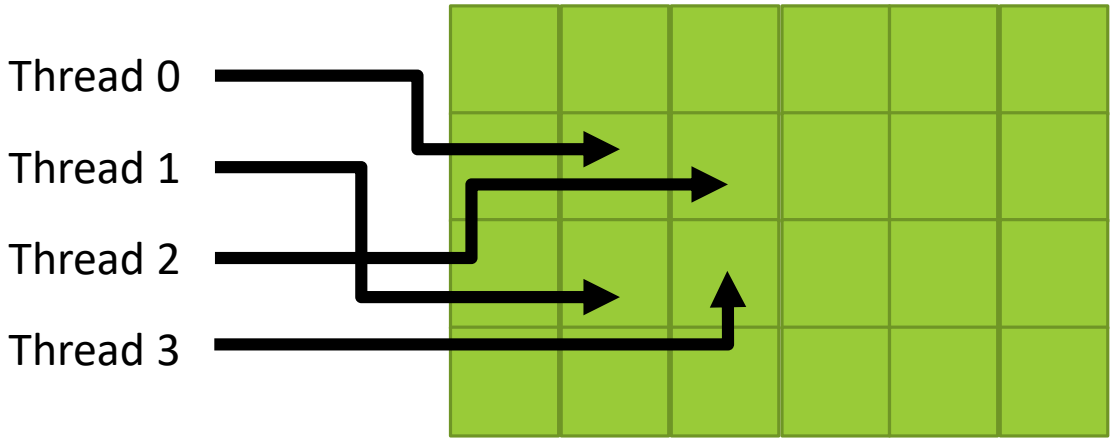
Posee su propio caché, el cual está optimizados para accesos cercanos en un espacio de dos dimensiones, por lo que está diseñada para aplicaciones gráficas.

Al igual que las demás memorias revisadas, el uso de texturas pueden mejorar el desempeño de la aplicación cuando ésta presenta ciertos patrones de acceso.

2D spatial locality

A pesar de que se puede crear una textura 1D, para obtener un real beneficio de su uso, es recomendable trabajar en dos dimensiones.

La “cercanía” mencionada corresponde a la cercanía de los datos no respecto a direcciones de memoria, sino en la estructura física que presentan.



Funcionamiento y uso

Las texturas presentan el mismo funcionamiento que la memoria constante, es decir, si el dato leído no está en cache, debe ser accedido desde la memoria VRAM. La diferencia radica, como ya vimos, en la optimización que presenta este caché.

Este tipo de memoria se utiliza cuando las hebras acceden a distintos datos en patrones que no pueden hacerse coalescentes, especialmente cuando hay cercanía en el espacio 2D representado.

Pasos

Para utilizar la memoria de texturas es necesario realizar una serie de pasos que no hemos visto hasta ahora:

- Declarar la referencia de textura
- Asignar la memoria de textura
- Enlazar la memoria a la referencia de textura
- Leer la memoria desde un kernel
- Desenlazar la memoria y la referencia
- Liberar la memoria

Texture reference

Una referencia de textura es un objeto que define el tipo de memoria de textura que se utilizará.

La referencia debe ser declarada en global scope mediante:

texture <type, dim> tex_ref;

donde *type* es el tipo de dato que la textura almacenará (limitado a enteros y flotantes) y *dim* es el número de dimensiones que presentará la textura (por defecto 1, pero también puede ser 2 o 3).

Por ejemplo, si quisiéramos una textura de dos dimensiones de valores enteros debemos declarar su referencia de esta forma:

texture <int, 2> tex_ref;

Asignación

La memoria de textura puede ser asignada en memoria global tal y como hemos hecho hasta el momento con *cudaMalloc()*.

Al hacerlo de esta forma, estamos limitados a utilizar texturas de una dimensión.

Otra forma de asignar la memoria de texturas es mediante el uso de arrays especiales de CUDA denominados *cudaArray*.

Éstos nos permiten utilizar texturas de más dimensiones y aprovechar su tipo de optimización.

Arrays de CUDA

Los array creados como *cudaArray* son un tipo especial de array optimizados para trabajar con texturas. Posee sus propias funciones para la mayoría de las acciones que comúnmente realizamos.

Para crear y asignar la memoria del array debemos ejecutar:

```
cudaArray *cuArray;
cudaMallocArray(&cuArray, &cD, ancho, alto);
```

donde *cD* es un objeto previamente creado llamado *cudaChannelFormatDesc* (define la configuración del array) y los parámetros *ancho* y *alto* describen el número de columnas y filas del array (*alto = 0* para arrays 1D).

Para liberar un array de CUDA debemos utilizar:

```
cudaFreeArray(cuArray);
```

Arrays de CUDA

Para definir el objeto *cudaChannelFormatDesc* debemos utilizar:

cudaChannelFormatDesc cD = cudaCreateChannelDesc(x, y, z, w, cudaChannelFormatKind);

donde *x*, *y*, *z* y *w* son la cantidad de bits en la dimensión respectiva y *cudaChannelFormatKind* puede tomar los valores *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned* y *cudaChannelFormatKindFloat* dependiendo del tipo de dato a utilizar.

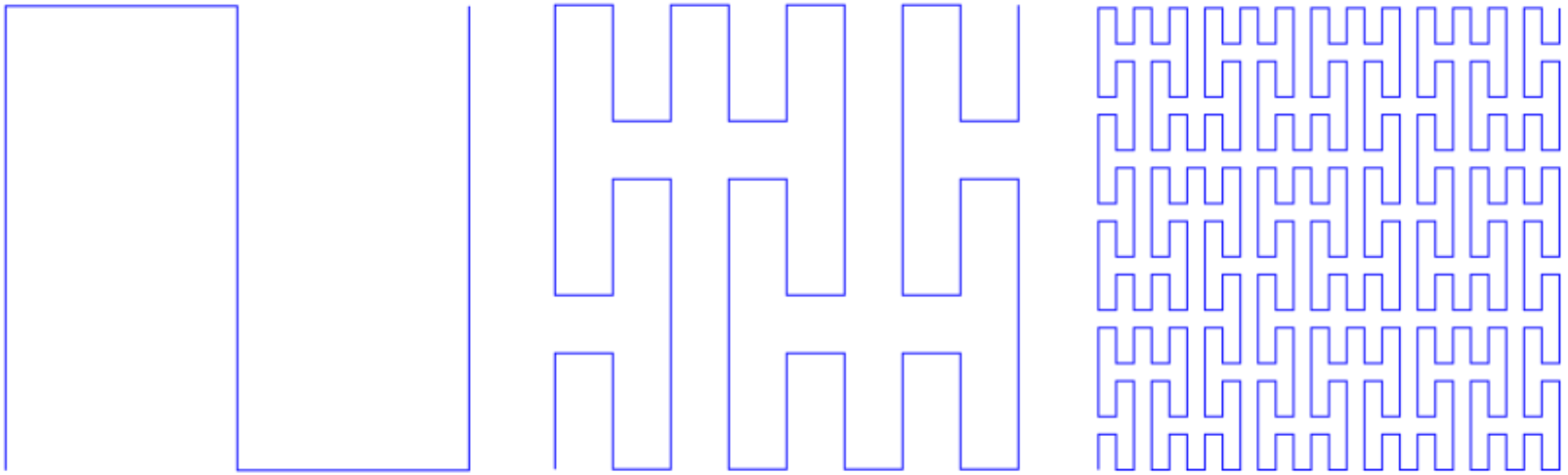
Por ejemplo, para crear un array de flotantes de precisión simple, debemos generar lo siguiente:

cudaChannelFormatDesc cD = cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);

Arrays de CUDA

Los datos copiados a un array de CUDA son automáticamente formateados a una curva de Peano (space-filling curve).

Este proceso es el causante de la optimización del caché relacionada a la cercanía 2D.



Enlace memoria-referencia

Enlazar la referencia de textura declarada con la memoria asignada nos permite acceder posteriormente a la memoria de textura mediante el uso de la referencia.

Si asignamos la memoria como un arreglo de memoria global, debemos enlazarla con la referencia mediante:

cudaBindTexture(offset, tex_ref, array, size);

Por otro lado, si decidimos utilizar arrays de CUDA, entonces debemos realizar el enlace mediante:

cudaBindTextureToArray(tex_ref, cuArray, cD);

Para desenlazar, sin importar la asignación utilizada, tenemos:

cudaUnbindTexture(tex_ref);

Leer texturas

Una vez que tengamos la referencia declarada, la memoria asignada y ambas enlazadas, al fin podemos realizar lecturas desde un kernel.

Esta acción también depende del tipo de asignación de memoria que hayamos utilizado. Si usamos arreglos de memoria global, las hebras leen datos mediante:

type data = tex1Dfetch(tex_ref, i);

donde *i* es la posición del elemento en el array.

Si utilizamos arrays de CUDA, debemos usar:

type data = tex2D(tex_ref, x, y);

donde *x* e *y* son la fila y la columna del dato respectivamente.

Ejemplo

```
texture<int, 1> tex_ref;
__global__ void kernel(int *dst, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N) dst[tId] = tex1Dfetch(tex_ref, tId);
}
int main(){
    int N = 10000, block_size = 256;
    int grid_size = (int)ceil((float)N / block_size);
    int *dst; cudaMalloc(&dst, N * sizeof(int));
    int *src; cudaMalloc(&src, N * sizeof(int));
    fill<<<grid_size, block_size>>>(src, N); // fill the input data
    cudaBindTexture(0, tex_ref, src, N * sizeof(int));
    cudaMemcpy(src, host_src, N*sizeof(int), cudaMemcpyHostToDevice);
    kernel<<<grid_size, block_size>>>(dst, N);
    cudaUnbindTexture(tex_ref);
    cudaFree(src); cudaFree(dst);
    return 0;
}
```

Ejemplo

```
texture<int, 2> tex_ref;
__global__ void kernel(int *dst, int N){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N * N) dst[tId] =
                        tex2D(tex_ref, tId%N, tId/N);
}
```

```
int main(){
    int N = 10000, block_size = 256;
    int grid_size = (int)ceil((float)(N * N) / block_size);
    int *dst; cudaMalloc(&dst, N * N * sizeof(int));
    int *src; cudaMalloc(&src, N * N * sizeof(int));
    fill<<<grid_size, block_size>>>(src, N); // fill the input data
    cudaArray *cuArray;
    cudaChannelFormatDesc channelDesc =
            cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindSigned);
    cudaMallocArray(&cuArray, &channelDesc, N, N);
    cudaMemcpyToArray(cuArray, 0, 0, src, N * N * sizeof(int),
                    cudaMemcpyDeviceToDevice);
    cudaBindTextureToArray(tex_ref, cuArray, channelDesc);
    kernel<<<grid_size, block_size>>>(dst, N);
    cudaUnbindTexture(tex_ref);
    cudaFreeArray(cuArray);
    cudaFree(src); cudaFree(dst);
    return 0;
}
```