La memoria unificada de CUDA

Curso de Deep Learning y CUDA

Titulaciones Propias UMA

Manuel Ujaldón

Catedrático de Universidad
Departamento de Arquitectura de Computadores
Universidad de Málaga

NVIDIA
DEEP LEARNING INSTITUTE
UNIVERSITY AMBASSADOR

Contenidos [18 diapositivas]

I. La vertiente hardware. [3]

II. La vertiente software. [5]

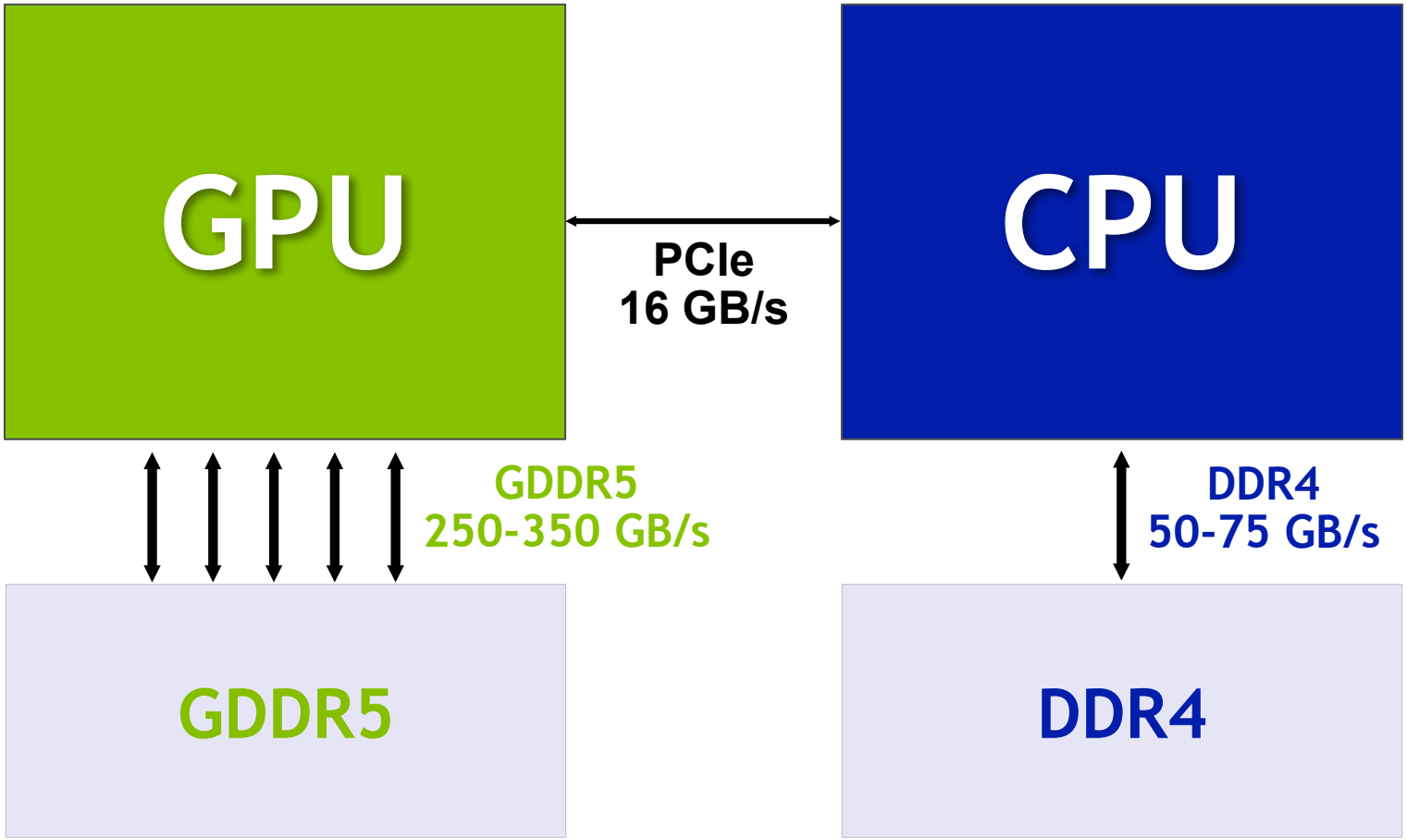
III. Ejemplos de programación. [9]

1. Restricciones de acceso. [2]
2. Incremento en paralelo. [1]
3. Ordenación de un vector. [1]
4. Clonando estructuras de datos dinámicas. [3]
5. Listas enlazadas. [2]

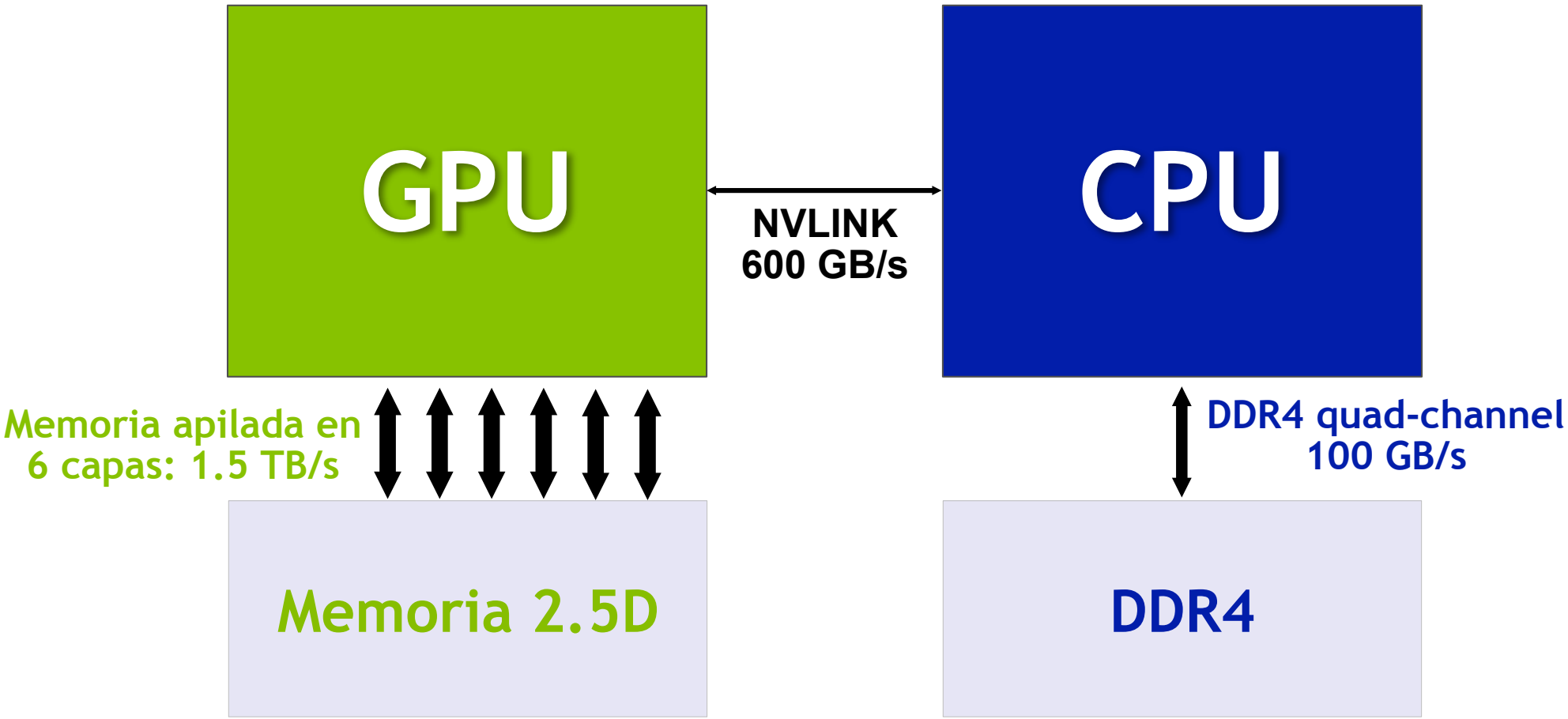
IV. Resumen. [1]

I. La vertiente hardware

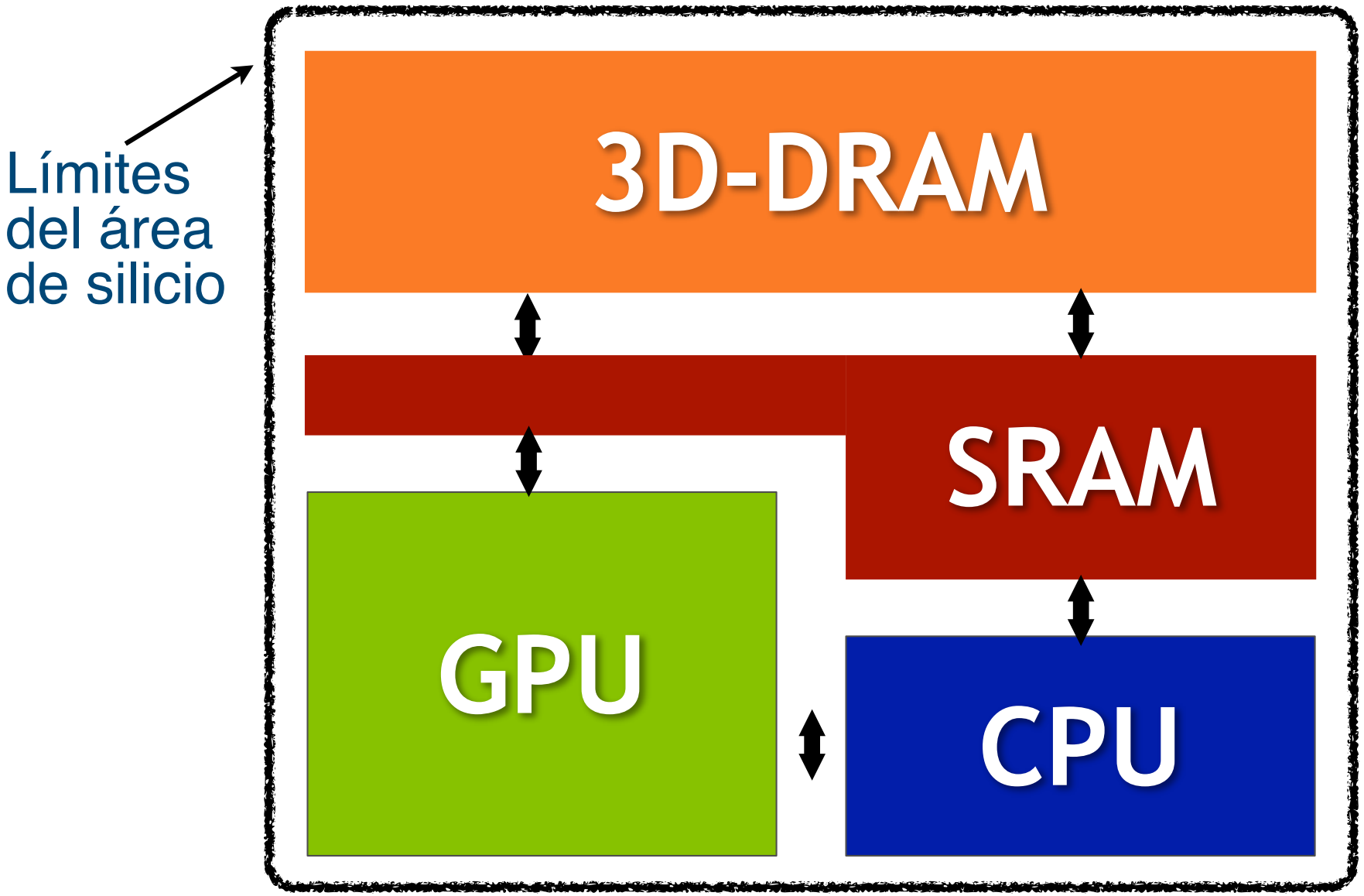
Evolución de la memoria.
Así estábamos en 2015



Así estamos en 2021 con la GPU Ampere A100



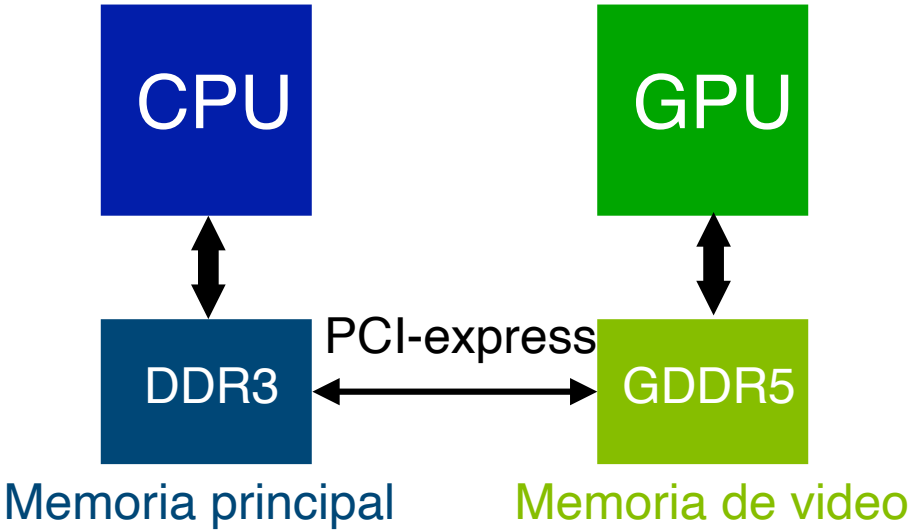
Predicción para 2025:
Todas las comunicaciones internas al chip



II. La vertiente software

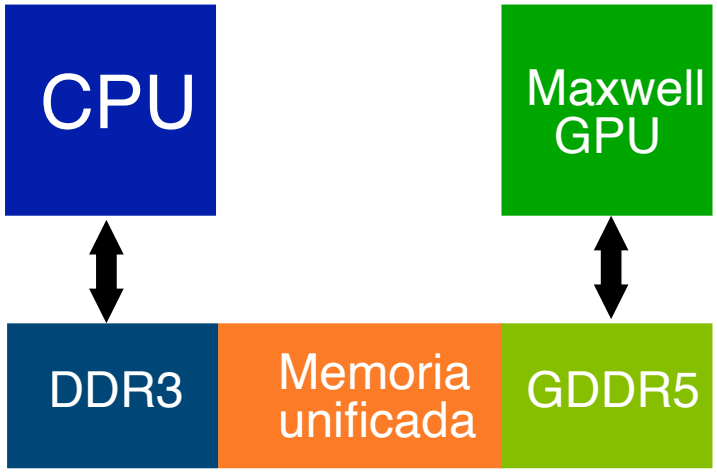
La idea: Tenemos que acostumbrar al programador a ver así a la memoria

CUDA 2007-2014



El viejo modelo software y hardware: Distintas memorias, prestaciones y espacio de direcciones.

CUDA en lo sucesivo



El nuevo API: Misma memoria, un solo espacio de direcciones.

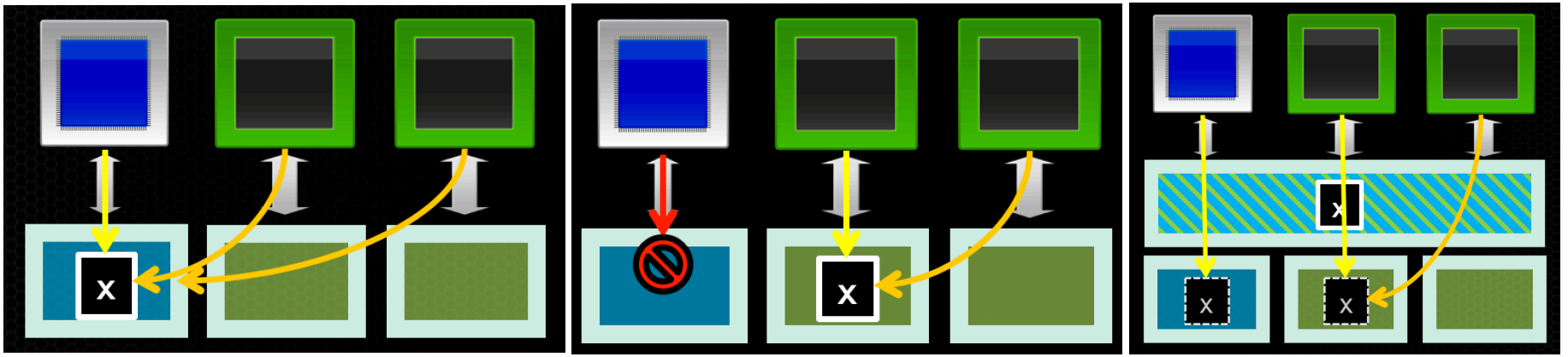
Rendimiento sensible a la proximidad de los datos.

Aportaciones de la memoria unificada

- Un modelo de programación y de memoria más simple:
  - Un puntero único a los datos, para acceder conjuntamente desde CPU y GPU.
  - Ya no hace falta utilizar `cudaMemcpy()`.
  - Simplifica enormemente la portabilidad del código.
- Mayor rendimiento a través de la localidad de los datos:
  - Migra los datos a la memoria del procesador que accede a ellos.
  - Garantiza la coherencia global.
  - Aún permite la optimización manual con `cudaMemcpyAsync()`.

Los tipos de memoria en CUDA

	Zero-Copy (pinned memory)	Unified Virtual Addressing	Unified Memory
Llamada CUDA	cudaMallocHost(&A, 4);	cudaMalloc(&A, 4);	cudaMallocManaged(&A, 4);
Alojada en	Mem. principal (DDR3)	Mem. video (GDDR5)	Ambas
Acceso local	CPU	La GPU de su tarjeta	La CPU y la GPU de su tarjeta
Acceso por PCI-e	Todas las GPUs	El resto de GPUs	El resto de GPUs
Otros rasgos	Evita paginación a disco	Prohibida desde la CPU	Migra al acceder desde CPU o GPU
Coherencia	En todo momento	Entre GPUs	Sólo con lanzar + sincronizar
Disponibilidad	CUDA 2.2	CUDA 1.0	CUDA 6.0



Novedades en el API de CUDA

- **`cudaMallocManaged(puntero,tamaño,flag)`**
  - Sustituto de `cudaMalloc(puntero,tamaño)` para alojar memoria.
  - El flag indica quién comparte el puntero con la GPU.
    - `cudaMemAttachHost`: Sólo la CPU.
    - `cudaMemAttachGlobal`: Adicionalmente, cualquier otra GPU.
  - Todas las operaciones válidas sobre la memoria de la GPU también son válidas sobre la memoria unificada.
- Nueva palabra clave: **`__managed__`**
  - Anotación de variable global que se combina con **`__device__`**.
  - Declara una variable de GPU migrable y de ámbito global.
  - Símbolo accesible tanto desde la CPU como desde la GPU.
- Nueva llamada: **`cudaStreamAttachMemAsync()`**
  - Gestiona concurrentemente las aplicaciones multi-hilo de la CPU.

Detalles técnicos

- La máxima cantidad de memoria unificada que puede alojarse es la **menor** de las memorias que tienen las GPUs.
- Aquella memoria unificada que sea tocada por la CPU debe **migrar** de regreso a la GPU antes de lanzar el kernel.
- La CPU no puede acceder a la memoria unificada mientras la GPU esté ejecutando, esto es, debemos llamar a `cudaDeviceSynchronize()` antes de permitir a la CPU que pueda acceder a la memoria unificada.
- La GPU tiene acceso **exclusivo** a la memoria unificada mientras se esté ejecutando un kernel, aunque éste no toque la memoria unificada (ver el primer ejemplo de la serie).

III. Ejemplos de programación

Ejemplo 1: Restricciones de acceso (2)

```
__device__ __managed__ int x, y = 2;          // Memoria unificada

__global__ void mykernel()                    // Territorio GPU
{
  x = 10;
}

int main()                                    // Territorio CPU
{
  mykernel <<<1,1>>> ();


  y = 20;          // ERROR: Acceso desde CPU concurrente con GPU
  return 0;
}
```

Ejemplo 1: Restricciones de acceso (2)

```
__device__ __managed__ int x, y = 2;          // Memoria unificada

__global__ void mykernel()                    // Territorio GPU
{
  x = 10;
}

int main()                                    // Territorio CPU
{
  mykernel <<<1,1>>> ();
  cudaDeviceSynchronize();                          // Solución
  // Ahora, la GPU está parada, el acceso a "y" no tiene riesgo
  y = 20;
  return 0;
}
```

Ejemplo 2: Incrementar un valor “b” a los N elementos de un vector “a”

Código CUDA pre-versión 6.0 **SIN** memoria unificada	Código CUDA post-versión 6.0 **CON** memoria unificada

`__global__ void incr (float *a, float b, int N)`
`{`
`  int idx = blockIdx.x * blockDim.x + threadIdx.x;`
`  if (idx < N)`
`  a[idx] = a[idx] + b;`
`}`
`void main()`
	`__global__ void incr (float *a, float b, int N)`
	`{`
	`  int idx = blockIdx.x * blockDim.x + threadIdx.x;`
	`  if (idx < N)`
	`  a[idx] = a[idx] + b;`
	`}`
	`void main()`
	`{`

`{`
`  unsigned int numBytes = N*sizeof(float);`
`  float* h_A = (float* ) malloc(numBytes);`
	`  float* m_A; cudaMallocManaged(&m_A, numBytes);`

`  float* d_A; cudaMalloc(&d_A, numBytes);`
`cudaMemcpy(d_A,h_A,numBytes,cudaMemcpyHostToDevice);`
`incr<<<N/blocksize,blocksize>>>(d_A, b, N);`
	`  incr<<<N/blocksize,blocksize>>>(m_A, b, N);`
	`  cudaDeviceSynchronize();`

`cudaMemcpy(h_A,d_A,numBytes,cudaMemcpyDeviceToHost);` `cudaFree(d_A);`	`  cudaFree(m_A);`
`  free(h_A);` `}`	`}`

Ejemplo 3: Ordenar un fichero de datos. Comparemos frente a las CPUs que usan C

Código para CPU (en C)

```
void sortfile (FILE *fp, int N)
{
  char *data;
  data = (char *) malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, comp);

  use_data(data);

  free(data);
}
```

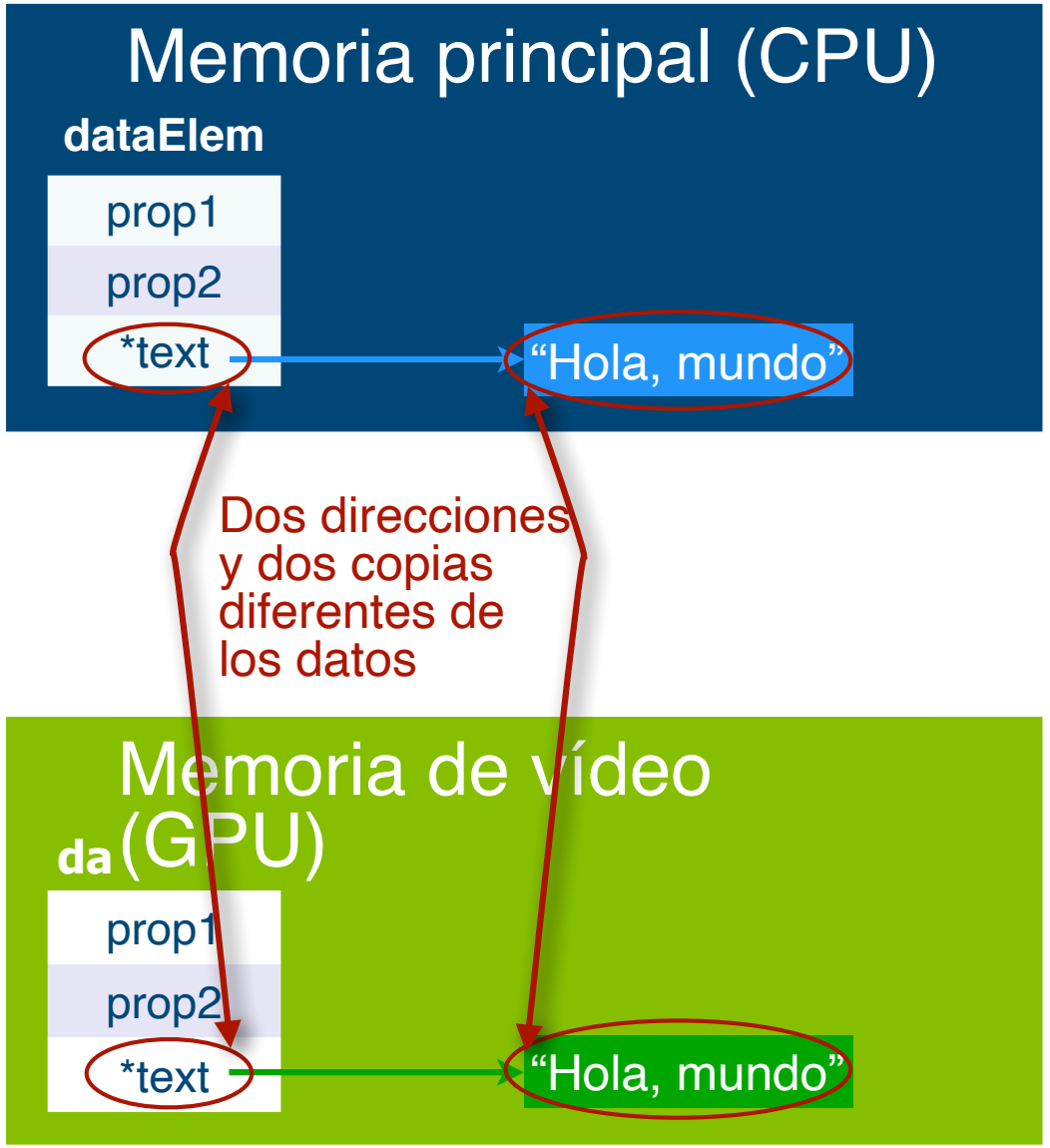
Código para GPU (a partir de CUDA 6.0)

```
void sortfile (FILE *fp, int N)
{
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data, N, 1, comp);
  cudaDeviceSynchronize();
  use_data(data);

  cudaFree(data);
}
```

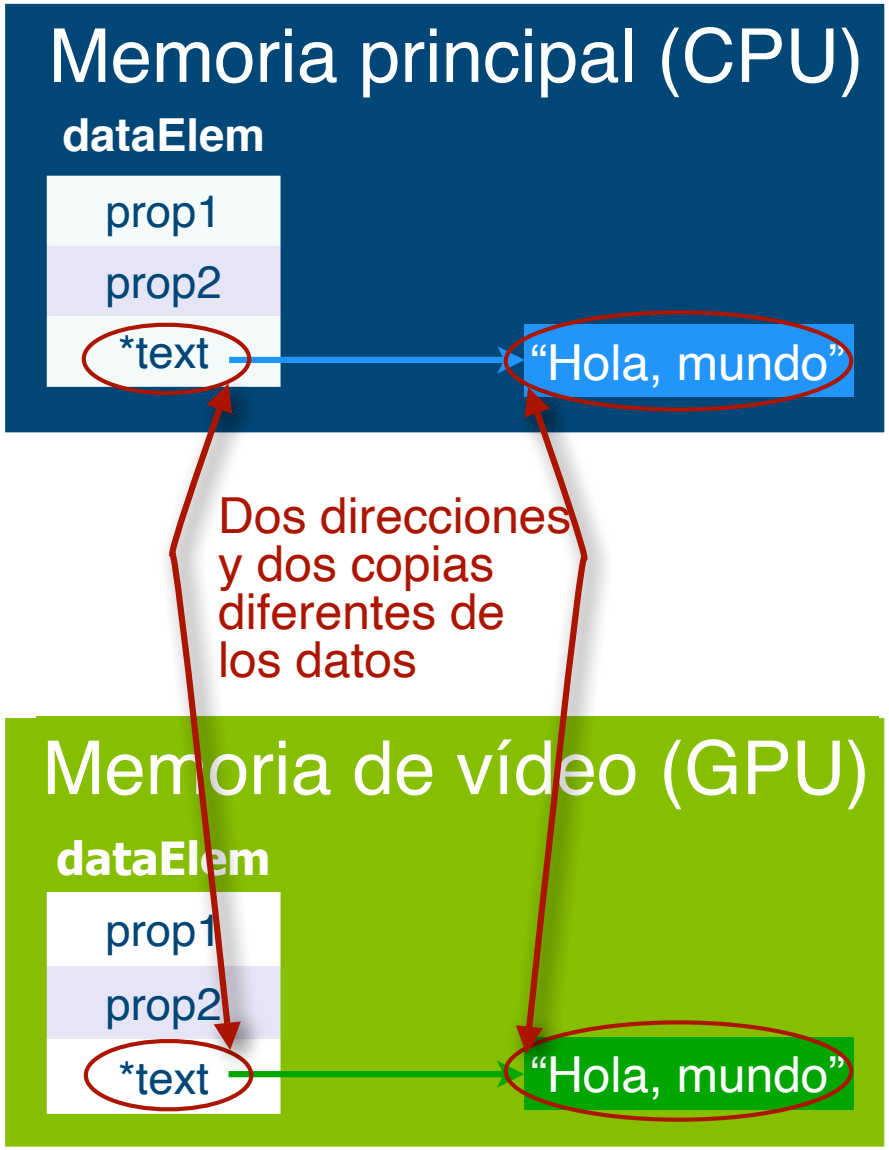
Ejemplo 4: Clonando estructuras dinámicas SIN memoria unificada

```
struct dataElem {
  int prop1;
  int prop2;
  char *text;
}
```

- Realizar copias sucesivas:
  - Debemos copiar la estructura y todos los contenidos a los que direcciona. Por eso C++ inventó el “copy constructor”.
  - CPU y GPU no pueden compartir una copia de sus datos (coherencia). Esto impide comparaciones tipo `memcpy`, `checksums` y demás.

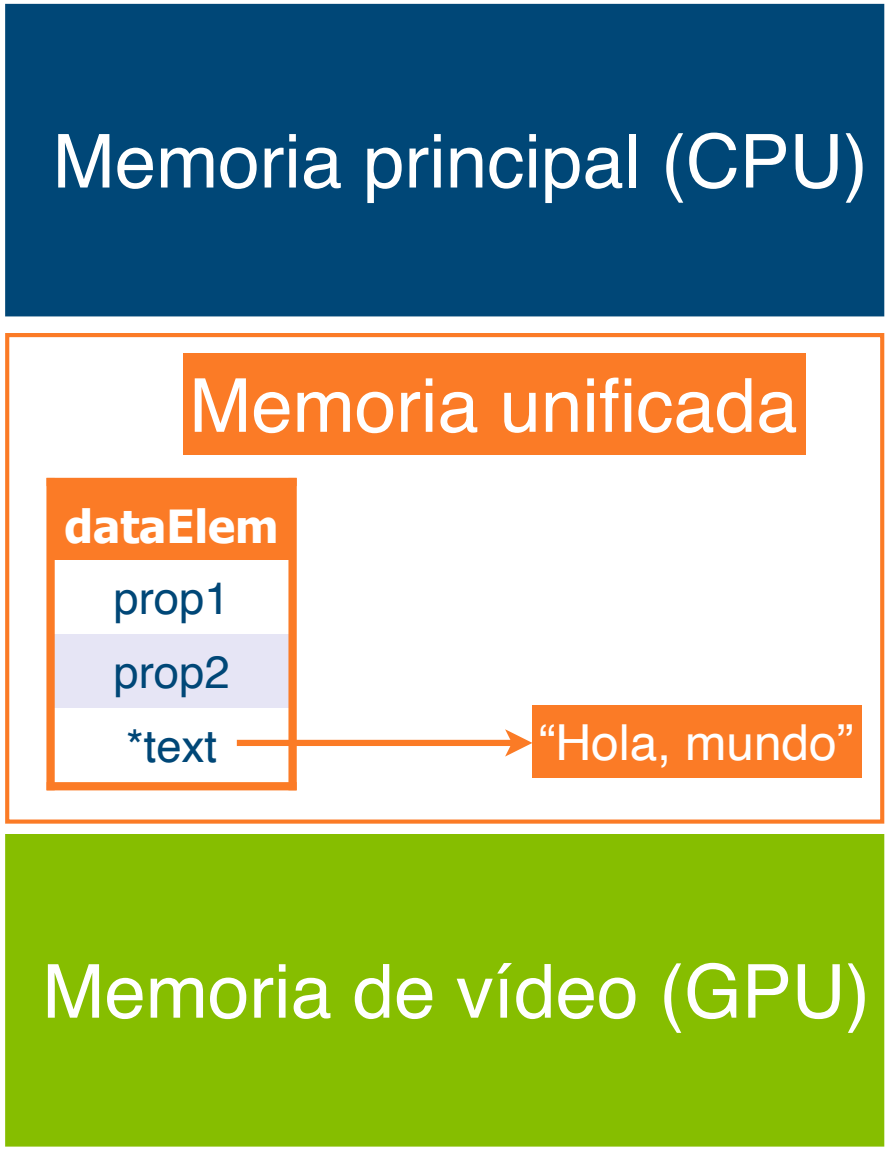


Clonando estructuras dinámicas SIN memoria unificada (2)




```
void launch(dataElem *elem) {
  dataElem *g_elem;
  char *g_text;

  int textlen = strlen(elem->text);

  // Aloja almacenamiento para struct y text
  cudaMalloc(&g_elem, sizeof(dataElem));
  cudaMalloc(&g_text, textlen);

  // Copia cada pieza por separado, incluyendo
un nuevo puntero en *text para la GPU
  cudaMemcpy(g_elem, elem, sizeof(dataElem));
  cudaMemcpy(g_text, elem->text, textlen);
  cudaMemcpy(&(g_elem->text), &g_text,
                                  sizeof(g_text));

  // Finalmente lanzamos el kernel, pero CPU y
GPU usan diferentes copias de "elem"
  kernel<<< ... >>>(g_elem);
}
```

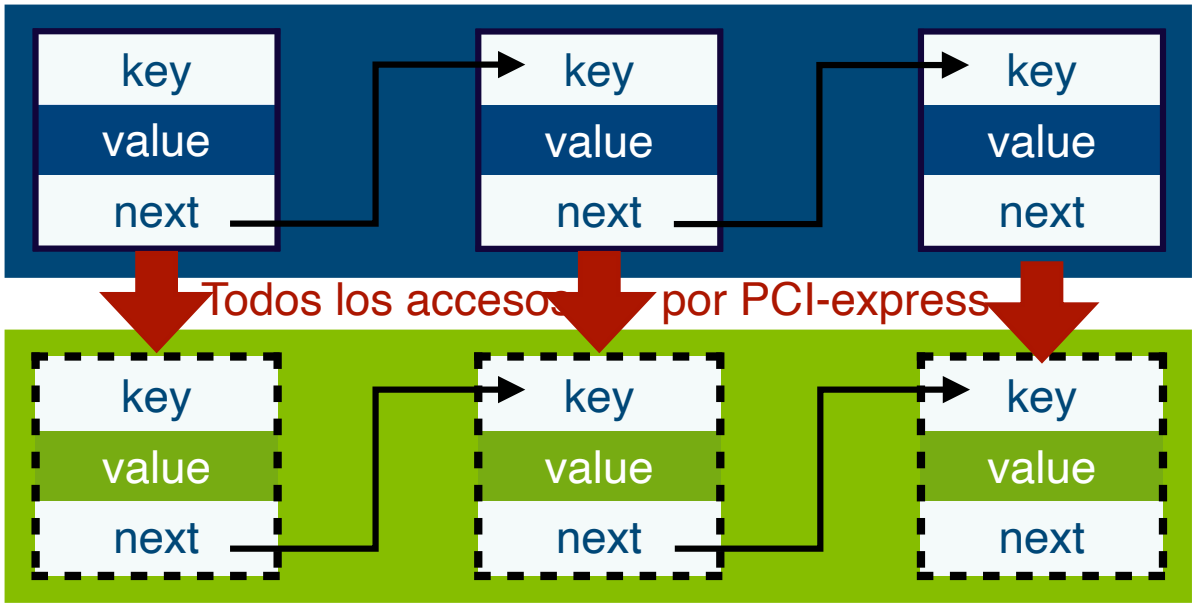
Clonando estructuras dinámicas CON memoria unificada




```
void launch(dataElem *elem)
{
    kernel<<< ... >>>(elem);
}
```

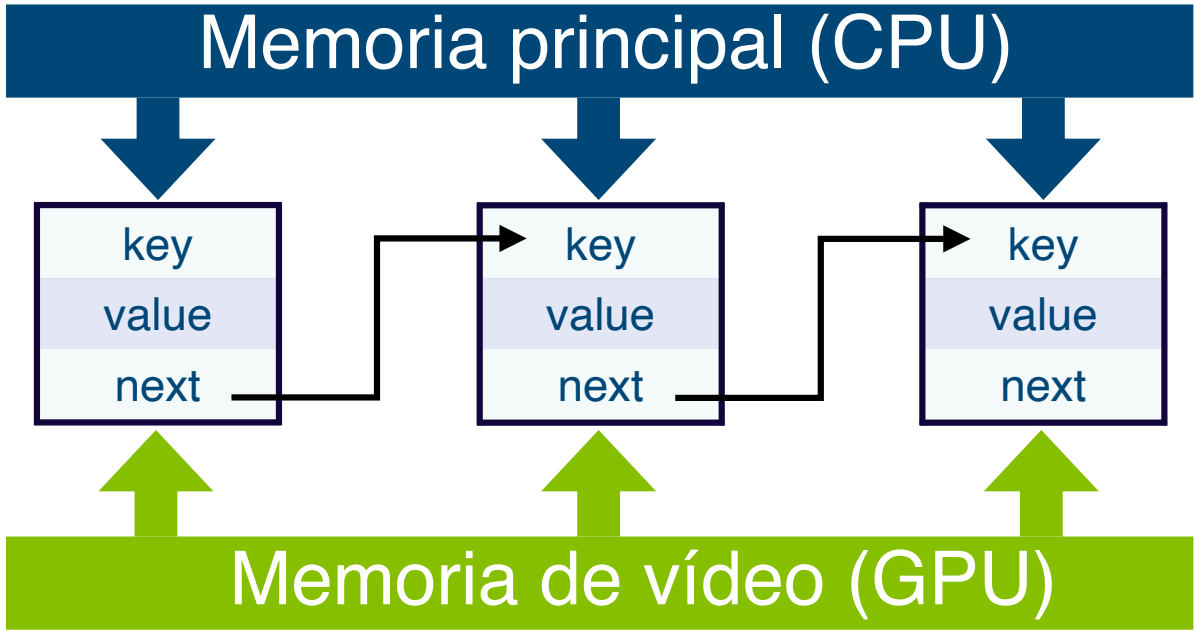
- Lo que queda invariable:
  - El movimiento de datos.
  - GPU usa una copia local de `text`.
- Lo que cambia:
  - El programador ve un solo puntero.
  - CPU y GPU acceden y referencian al mismo objeto.
  - Existe coherencia en memoria.
- Para pasar por referencia y por valor se necesita usar C++.

Ejemplo 5: Listas enlazadas



- Casi imposible de implementar con el API original de CUDA.
- La solución menos mala es utilizar memoria pinned:
  - Los punteros son globales, al igual que con memoria unificada.
  - Pero el rendimiento es bajo: La GPU padece los accesos por PCI-e.
  - La latencia de la GPU es muy alta, lo que resulta crítico para las listas enlazadas debido al usual recorrido encadenado de punteros.

Listas enlazadas con memoria unificada



- Se pueden pasar elementos entre la CPU y la GPU.
  - No hace falta mover datos entre la CPU y la GPU.
- Se pueden insertar y borrar elementos desde CPU o GPU.
  - Pero el programa debe prevenir condiciones de carrera (los datos son coherentes entre CPU y GPU solo si se lanza y sincroniza).

Memoria unificada: Resumen

- `cudaMallocManaged()` reemplaza a `cudaMalloc()`.
  - `cudaMemcpy()` es ahora opcional.
- Simplifica enormemente la portabilidad de código.
  - Menos gestión de la memoria en el lado del host.
- Permite compartir estructuras de datos entre CPU y GPU
  - Un mismo puntero al dato. No hay que clonar su estructura.
- Potente mecanismo en lenguajes de alto nivel como C++.