Escuela de Computación de Alto Rendimiento 2014

# Comunicación y múltiples GPU

Carlos Bederián
IFEG – CONICET
GPGPU Computing Group – FaMAF – UNC

CONICET

NVIDIA CUDA TEACHING CENTER

NVIDIA CUDA RESEARCH CENTER

UNC Universidad Nacional de Córdoba

# Streams - Motivación

- La GPU queda muy “lejos” del resto del sistema
  - PCI Express 3.0 16X: **16 GBps, bidireccional**
  - Necesidad de ocultar esta latencia
- La GPU tiene poca memoria
  - Necesidad de subir→procesar→bajar partes de sistemas grandes
    - Si el kernel es computacionalmente intensivo (ej: DGEMM), puede operarse sin pérdida de performance

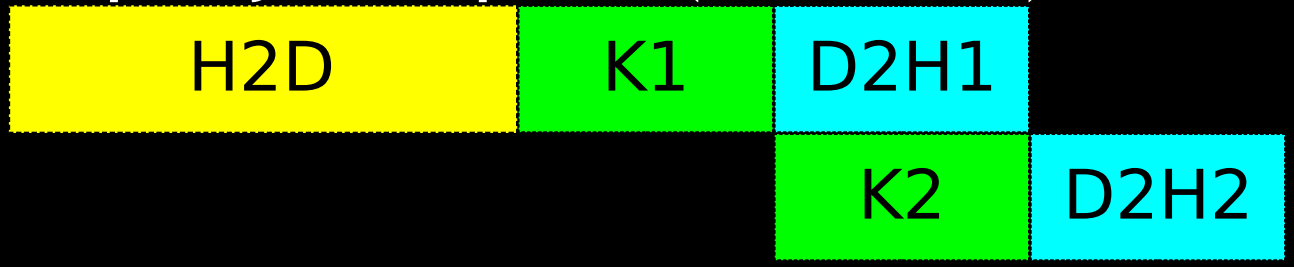
# Streams

- Modelo de concurrencia dentro de la GPU
- Colas de tareas que se ejecutan secuencialmente en el orden que se despachan
- Todos los streams se ejecutan concurrentemente (si pueden)
- Son una abstracción del hardware subyacente
  - CC <3.5: 1 cola de kernels, 1 o 2 colas de copias
  - CC 3.5: HyperQ – 32 tareas independientes

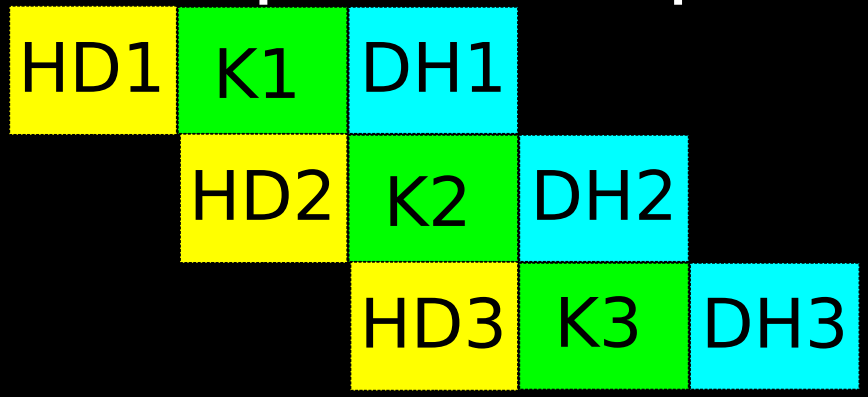
# Ocultamiento de latencia

Secuencial


Copia y cómputo (GeForce)




Dos copias + Cómputo (Tesla)



# Streams – Motivación (cont.)

- Necesidad de concurrencia entre kernels dentro de una GPU
  - Los últimos bloques de un kernel dejan SM libres
    - Barrera implícita
    - Podemos correr cosas si explicitamos paralelismo
  - A veces una GPU tiene demasiados recursos de ejecución para correr un único kernel
    - Queremos poder correr múltiples kernels en paralelo

# Streams – Creación/destrucción

```
cudaError_t cudaStreamCreate(cudaStream_t *s)
```

- Crea un nuevo stream y lo devuelve en *s
- Devuelve un `cudaError_t` como todo CUDA
  - **¡Verificar el resultado!** (nunca está de más recordarlo)
- Extras: `cudaStreamCreateWithPriority`, `cudaStreamCreateWithFlags`

```
cudaStreamDestroy(cudaStream_t stream)
```

- Espera que stream termine su trabajo y lo destruye.

# Streams – Ejecución

- Encolar un kernel en un stream:
  `kernel <<<grid, block, shared, stream>>> (...)`
- `shared`: Memoria compartida definida en tiempo de ejecución (0 bytes por defecto)
  - En el kernel: `extern __shared__ T buf[];`
- `stream`: Stream donde encolar el kernel.
  - Por defecto: Stream 0 (“null stream”)
    - Creado automáticamente
    - Sincroniza la GPU al completar cada tarea
      - **OJO: ¡No sincroniza con el host!**

# Streams – Bibliotecas

- Encolan sus kernels en el stream que se les configure

- CUFFT

```
cufftResult cufftSetStream(cufftHandle plan,
                           cudaStream_t stream);
```

- CUBLAS

```
cublasStatus_t cublasSetStream(
    cublasHandle_t handle, cudaStream_t stream);
```

- CUSPARSE

```
cusparseStatus_t cusparseSetStream(
                           cusparseHandle_t h,
                           cudaStream_t str);
```

# Streams – Comunicación

- Encolar un memset:
  `cudaMemset*Async(... , cudaStream_t stream)`

- Encolar una transferencia:
  `cudaMemcpy*Async(... , cudaStream_t stream)`

- **OJO**: La memoria del host debe ser **pinned** para que funcione

# Pinned memory

- Las copias asíncronas son manejadas por un DMA engine en la GPU
  - ¡Pero el sistema operativo puede mudar las páginas de memoria!
- CUDA provee memoria no paginable en el host:

```
cudaMallocHost (void ** ptr, size_t size)
cudaFreeHost (void * ptr)
```

  - Uso similar a `cudaMalloc` y `cudaFree` pero para memoria del host
  - Copias más rápidas

# Streams – Sincronización

- Esperar todo el trabajo actual en la GPU:
  `cudaDeviceSynchronize()`

- Esperar que termine un stream:
  `cudaStreamSynchronize(cudaStream_t s)`

- Consultar si un stream terminó (sin bloquear):
  `cudaError_t cudaStreamQuery(cudaStream_t s)`
  - ¡No se puede usar con las macros habituales!

# Streams – Ejemplo

```
cudaStream_t stream[3];
for (int i = 0; i < 3; ++i) {
  cudaStreamCreate(&stream[i]);  // crear streams
  cudaMallocHost(&h_b[i], size); // pinned memory
}
for (int i = 0; i < 3; ++i) {
  kernel_A <<<grid, block, 0, stream[i]>>> (…);
  kernel_B <<<grid, block, 0, stream[i]>>> (…);
  kernel_C <<<grid, block, 0, stream[i]>>> (…);
  cudaMemcpyAsync(h_b[i], d_b[i], …, stream[i]);
}
cudaDeviceSynchronize();
// hacer algo con h_b
for (int i = 0; i < 3; ++i) {
 cudaStreamDestroy(stream[i]);
 cudaFreeHost(h_b[i]);
}
```

# Streams – Callbacks

- Encolar ejecución de código en el host en un stream:

```
cudaStreamAddCallback(cudaStream_t s,
                      cudaStreamCallback_t cb, void* data, 0);
```

  - cudaStreamCallback_t es de tipo:

```
void(CUDART_CB* cudaStreamCallback_t(
          cudaStream_t stream,
          cudaError_t status,
          void* userData)
```

  - El callback en ejecución bloquea el stream
  - **El callback no puede tener llamadas a CUDA**

# Streams – Callbacks (ejemplo)

```
void CUDART_CB imprimir(cudaStream_t stream,
                        cudaError_t status,
                        void * data) {
  printf("Terminó de copiar\n");
}

cudaMemcpyAsync(dst, src, sz,
                cudaMemcpyDeviceToHost, s1);
cudaStreamAddCallback(s1, &imprimir, NULL, 0);
```

# Streams – Eventos

- Esperar que termine un stream o todo el trabajo encolado en la GPU a veces no es suficientemente fino

- Los eventos se introducen para cubrir esta necesidad
  - Marcadores que se ponen entre operaciones de un stream
  - Funciones de sincronización de eventos

# Eventos

- Creación

```
cudaEvent_t event;
cudaEventCreate(&event);
```

- Destrucción

```
cudaEventDestroy(event);
```

- Inserción en un stream

```
cudaEventRecord(event, stream);
```

# Eventos (cont.)

- Ver si ocurrió un evento

```
cudaError_t status = cudaEventQuery(event);
```

- Esperar que ocurra un evento

```
cudaEventSynchronize(event);
```

- Hacer que un stream espere que se dé un evento antes de seguir:

```
cudaStreamWaitEvent(stream, event, 0);
```

  - Esto ocurre sin intervención del host
  - Sincronización entre distintos streams
    - Grafo dirigido de dependencias

# Streams – Cosas a evitar

- Stream nulo
  - Sincroniza el device después de cada operación
  - Sincroniza el host después de cada operación, salvo algunas excepciones
- Sincronización implícita
  - `cudaMallocHost / cudaHostAlloc`
  - `cudaMalloc`
  - `cudaMemcpy* / cudaMemset* (no Async)`
  - …
- **OJO: Estamos acostumbrados a la falta de concurrencia**

# MultiGPU - Motivación

- No desperdiciar recursos
  - Placas con múltiples GPU (Tesla K10, GTX 590/690)

- Encarar problemas más grandes
  - En tiempo de ejecución
  - En tamaño del sistema

# MultiGPU - Motivación (cont.)

- Sequoia (BG/Q, 16,7 PFLOPS, 7,9MW): ¡HPL toma el equivalente de 100 barriles de petróleo!
- Utilización efectiva de clusters heterogéneos
- En Green 500, Junio 2014:
  - Kepler + Xeon E5: 2.2 – 4.4 GFLOPS/W
    - ¡15 primeros puestos!
  - Xeon Phi: 1.1 – 2.5 GFLOPS/W
  - POWER: 2.1 – 2.3 GFLOPS/W

# Formas de usar múltiples GPU

- Un proceso, múltiples GPU
  - Sólo funciona en un nodo
    - Un hilo controla todas las GPU
    - División de las GPU entre hilos
- Múltiples procesos
  - Pasaje de mensajes entre procesos
  - Comunicación intra-nodo o inter-nodo

# Contextos

- Hay que crear un contexto CPU-GPU antes de hacer cualquier cosa en una GPU
  - La primer operación que cambia el estado del device actual crea un contexto de manera transparente
- Límite de contextos por GPU determinados por el administrador usando `nvidia-smi`
- Para dejar de usar un device, destruímos su contexto en el proceso:

  `cudaDeviceReset()`

# Devices

- Ver cantidad de devices presentes:

```
cudaGetDeviceCount (int* count)
```

- Ver características de un device particular:

```
cudaGetDeviceProperties(cudaDeviceProp* p,
                        int device)
```

  - Están ordenados por velocidad (¿precio?)

- Cambiar de device:

```
cudaSetDevice (int device)
```

# Uso

- Un proceso, un hilo:
  1. Para cada device:
     - Cambiar al device
     - Encolar operaciones
  2. Para cada device:
     - Cambiar al device
     - Sincronizar
- Un proceso, muchos hilos:
  - Cada hilo toma un device

# Comunicación entre devices

- Device → Host → Otro device
  - Funciona en todos lados
  - Overhead innecesario

```
cudaSetDevice(0);
cudaMemcpy*(h, d0, sz, cudaMemcpyDeviceToHost);
cudaSetDevice(1);
cudaMemcpy*(d1, h, sz, cudaMemcpyHostToDevice);
```

- Atajo:

```
cudaMemcpy*Peer*(d1, 1, d0, 0, sz);
```

# Pero...

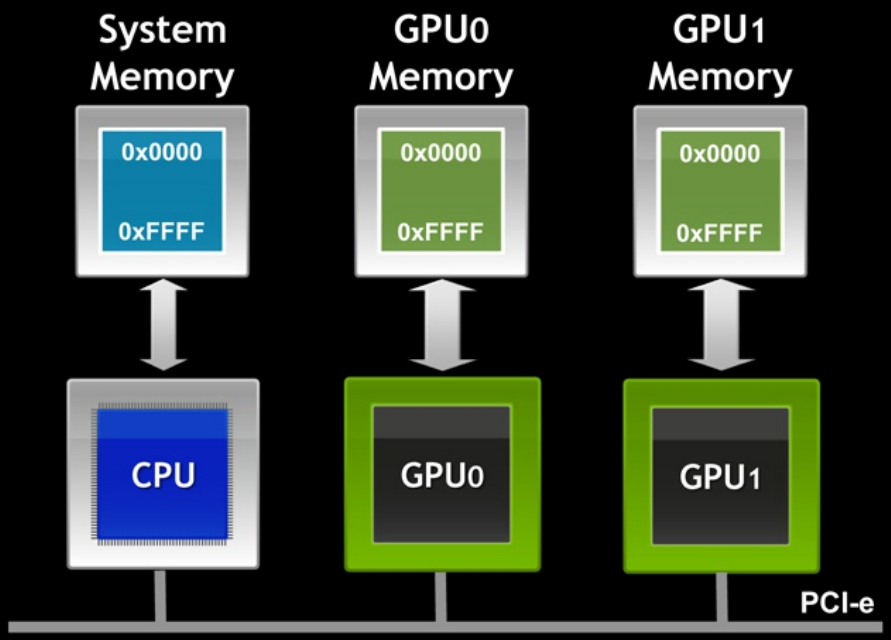
- Llevar rastro de todo esto es engorroso
- ¡El driver ya sabe dónde queda cada cosa!
  - Si sólo tuviera suficientes direcciones...

# Unified Virtual Addressing

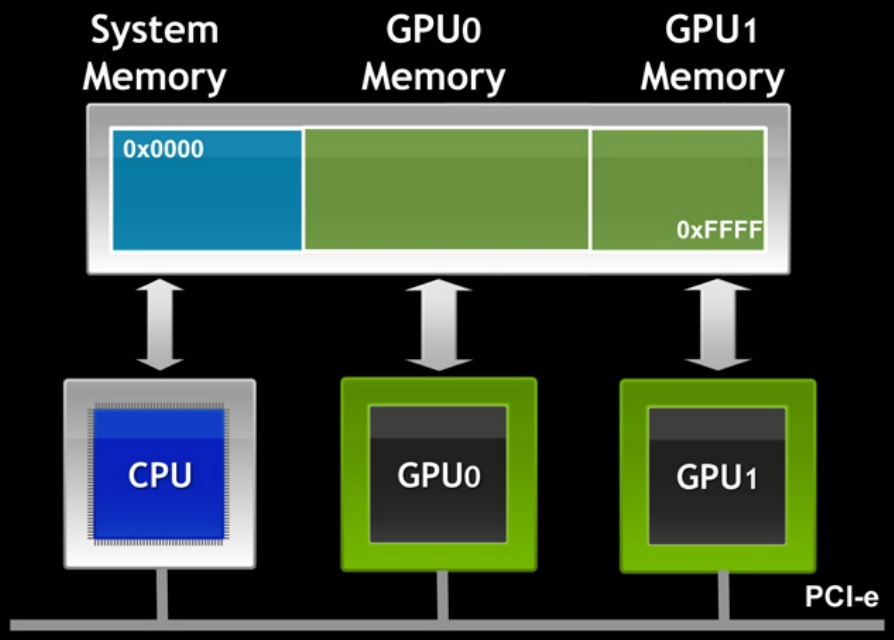
## Unified Virtual Addressing

***Easier to Program with Single Address Space***

***No UVA: Multiple Memory Spaces***




***UVA : Single Address Space***



# Unified Virtual Addressing

- Requiere:
  - CUDA 4.0
  - Fermi
  - Sistema de operativo y aplicación de 64 bits
- En una aplicación de 64 bits hay espacio de direcciones de sobra
- UVA: El driver mapea toda la memoria de las GPU al mismo espacio de direcciones del proceso y lleva rastro de a quién le pertenece qué cosa.

# Comunicación con UVA

```
cudaMemcpy*(..., cudaMemcpyDefault)
```

- Copia de **cualquier lado** a **cualquier lado** usando UVA (H2H, H2D, D2H, D2D, P2P*)

# Acceso directo entre GPU

```
cudaDeviceEnablePeerAccess (int peerDevice, 0)
```

- Permite que la GPU activa acceda a la memoria de otra GPU vía PCIe
  - Podemos llamar un kernel con un puntero de otra GPU
  - Datos cacheados en L2 de la GPU destino
    - No hay coherencia

- **OJO: Establecer el acceso en ambos sentidos**

# Memcpy directo entre GPU

1. Activar peer access

   (Sin peer access copia a través del host)

2. `cudaMemcpy(dst, src, sz, cudaMemcpyDefault);`

# Sincronización multi-GPU

- No hay muchas opciones
  - Sincronizar desde el host con `cudaDeviceSynchronize` en todas las GPU
  - Sincronización automática con `cudaStreamWaitEvent`

# Comunicación en clusters

- MPI
- Necesidad de comunicar datos por la red

  GPU → Host memory → Send buffer → Red

  → Recv buffer → Host memory → GPU (!!!)
- Las redes lentas limitan la escalabilidad
  - Ethernet 10GbE, 40GbE
  - Infiniband DDR (16 Gbps), QDR (32 Gbps), FDR10 (41 Gbps), FDR (55 Gbps), EDR (100 Gbps)
    - ¡PCI Express 3.0 16X (16 GBps) ya es cuello de botella!

# CUDA con MPI

- Paralelización: 1 GPU por proceso
- Código:

```
cudaMemcpy(h_send, d_send, …);
MPI_Send(h_send, …);
MPI_Recv(h_recv, …);
cudaMemcpy(d_recv, h_recv, …);
```

- Compilación (versión menos problemática)
  1. Separar código CUDA, compilarlo con `nvcc`
  2. Compilar código host y MPI con `mpicc`
  3. Linkear con `mpicc`, agregando libs de CUDA
     - Generalmente `-L/usr/local/cuda/lib64 -lcudart`

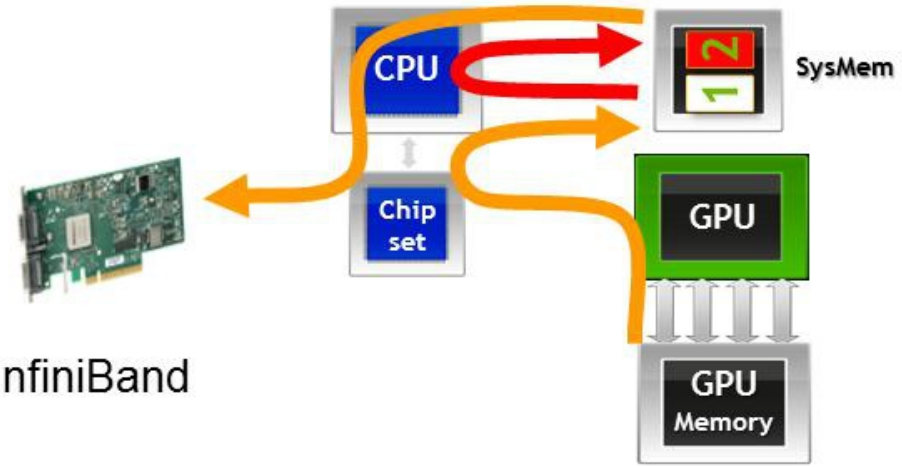
# Quitando copias

- GPUDirect v1 (2010)
  - Colaboración entre NVIDIA y Mellanox

**Without GPUDirect**

**Same data copied three times:**

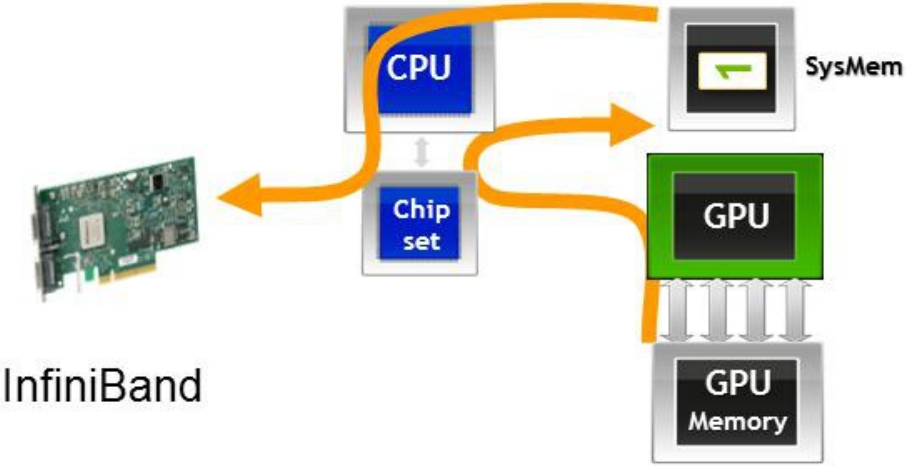
1. **GPU writes to pinned sysmem1**
2. **CPU copies from sysmem1 to sysmem2**
3. **InfiniBand driver copies from sysmem2**




**With GPUDirect**

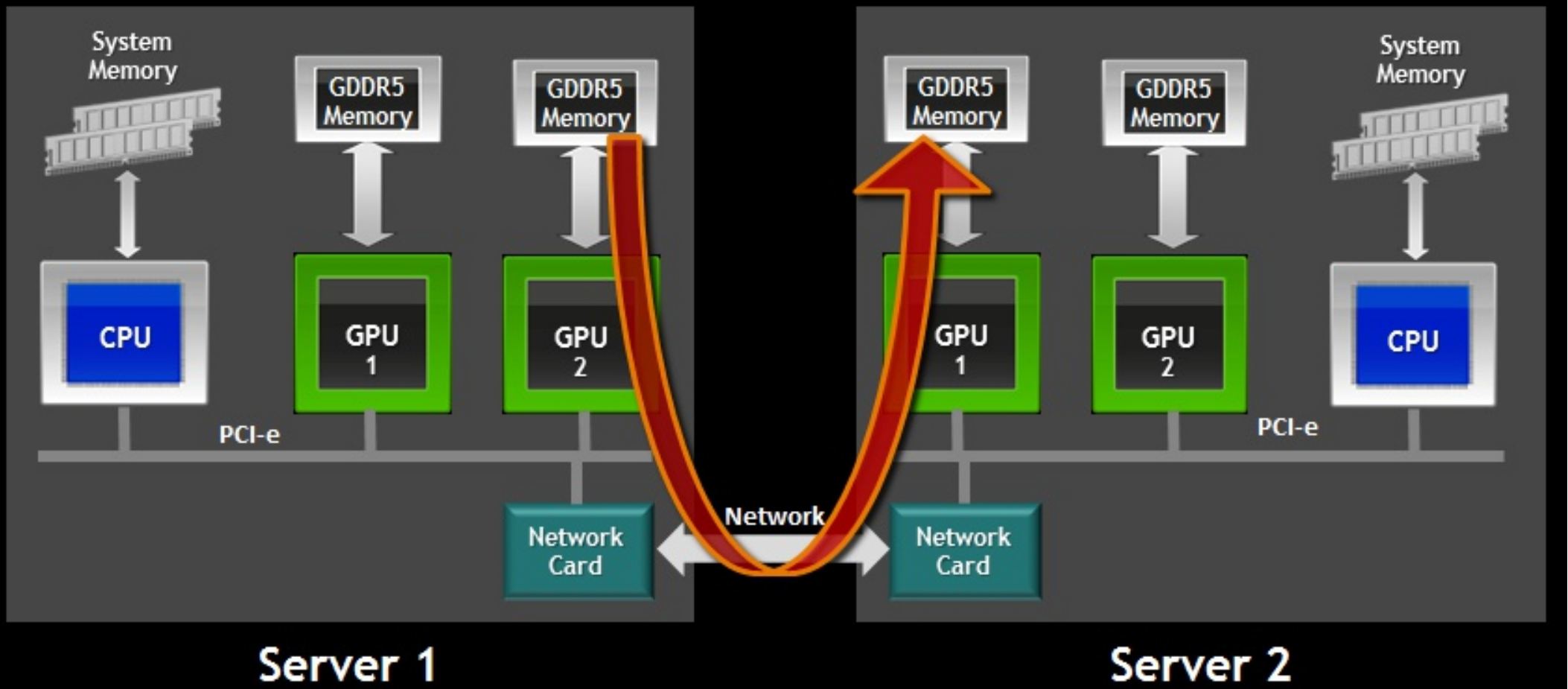
**Data only copied twice**

**Sharing pinned system memory makes sysmem-to-sysmem copy unnecessary**

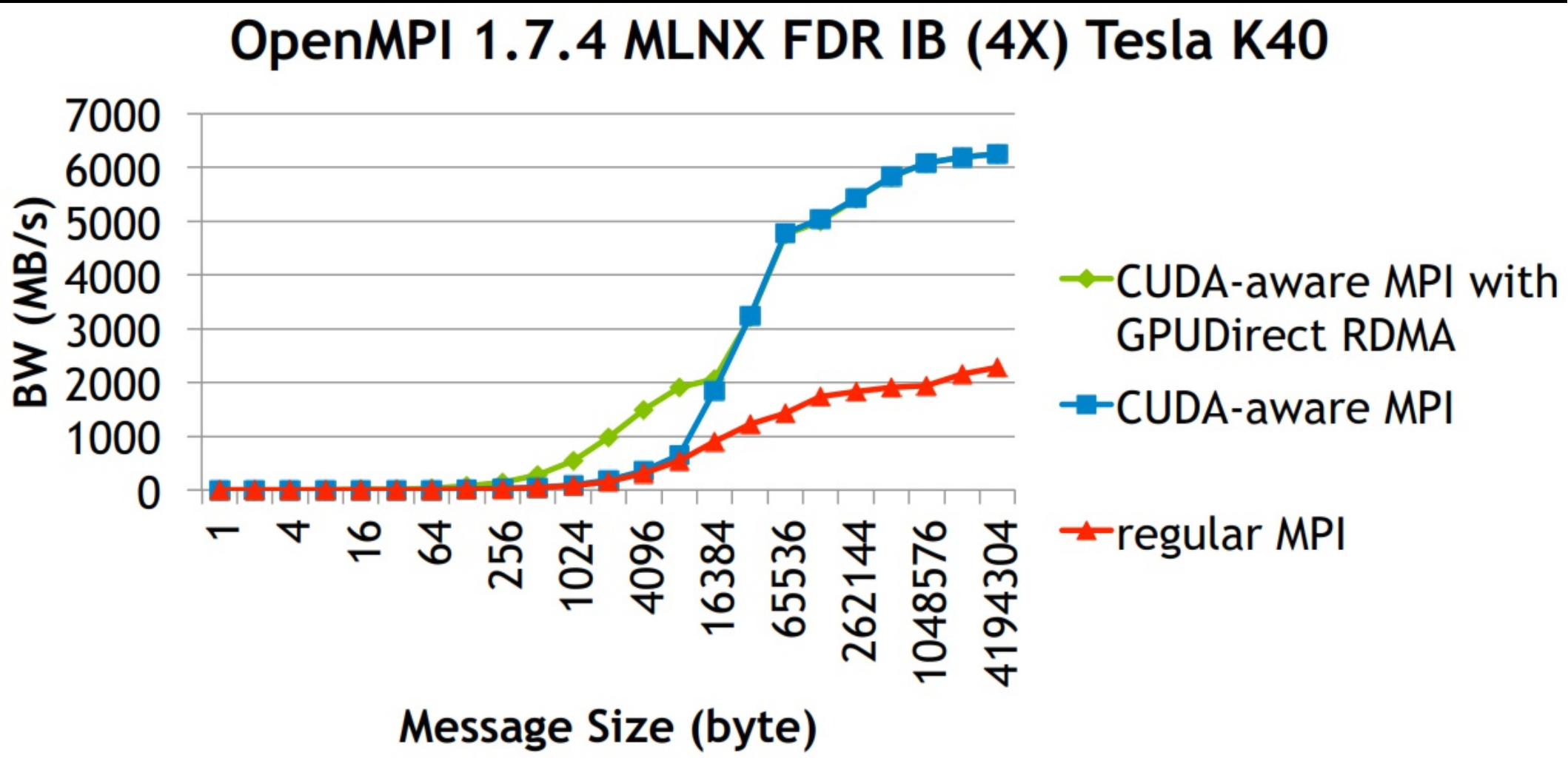


# Sigue habiendo intermediarios

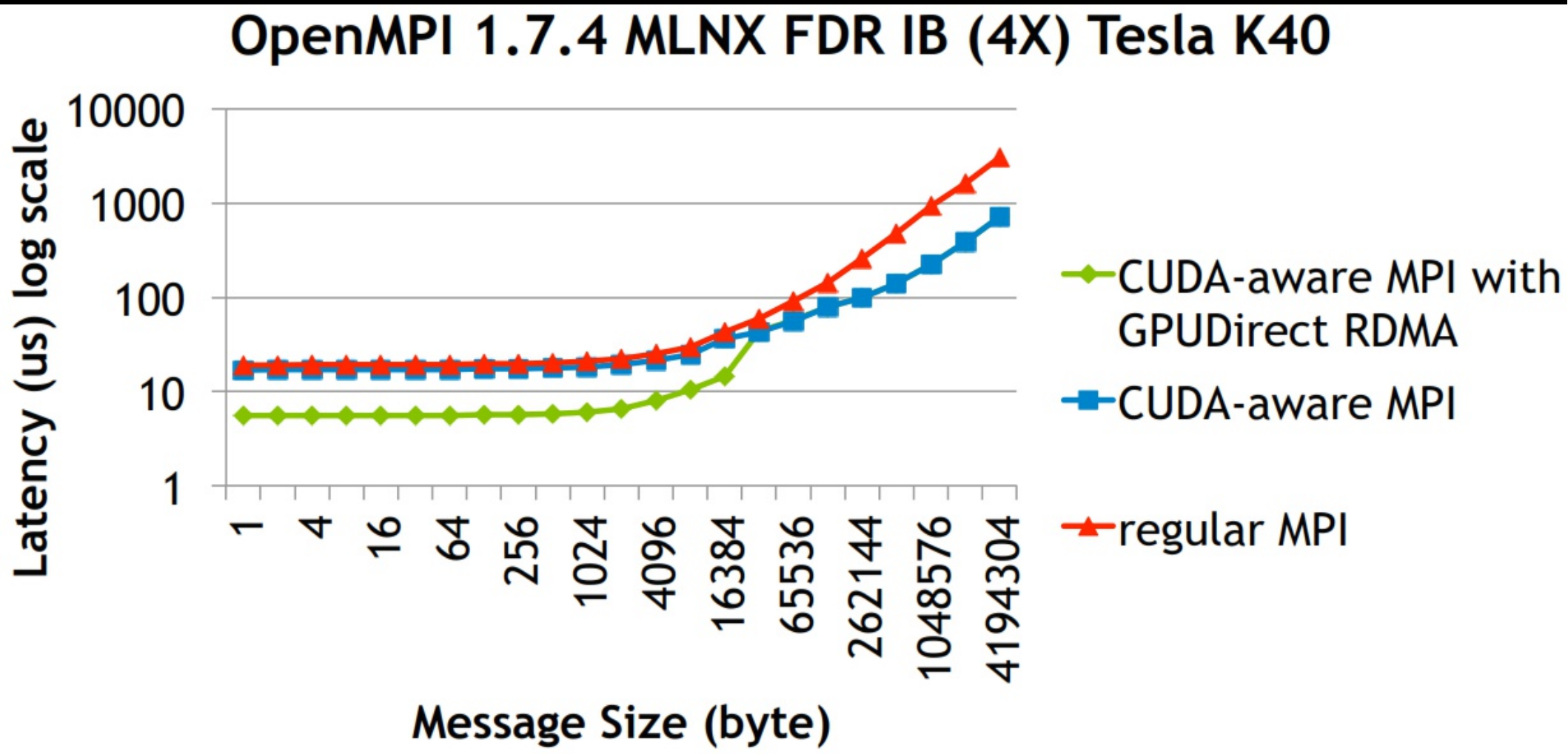
- GPUDirect RDMA (2014)
  - Kepler + Mellanox ConnectX-3 / ConnectX-IB



# Ancho de banda



# Latencia


OpenMPI 1.7.4 MLNX FDR IB (4X) Tesla K40
Latency (us) log scale
10000
1000
100
10
1
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
Message Size (byte)
CUDA-aware MPI with GPUDirect RDMA
CUDA-aware MPI
regular MPI

# ¿Cómo los uso?

- Implementaciones de MPI CUDA-aware

  (con código aportado por NVIDIA y Mellanox)
  - MVAPICH2 / MVAPICH2-GDR
  - OpenMPI

- Requieren compilación y configuración adicional
  - Soborne a su administrador de cluster más cercano

# CUDA-Aware MPI

- Simple: pasar punteros en GPU a MPI
  - La implementación detecta y actúa

```
MPI_Send(d_send, …);
MPI_Recv(d_recv, …);
MPI_Reduce(d_value, d_result, …);
```

...etc

- Todo **debería** funcionar
  - Las implementaciones mejoran con cada versión
  - Hoy: maduras (MVAPICH2 2.0, OpenMPI 1.8.3)

# Código MPI heredado

- Implementaciones que dividen en más procesos de los que querríamos
  - A veces no podemos convertirlos en MPI+OpenMP
  - Muchos paquetes tienen este problema
    - HACC, MP2C, Quantum Espresso, VASP…
- Solución: CUDA Multi-Process Service
  - Multiplexa varios procesos a una GPU
  - Hasta 16 procesos
  - Desde Kepler
  - Configuración depende del entorno

# Se terminó

- ¿Consultas?