Tema Seminario

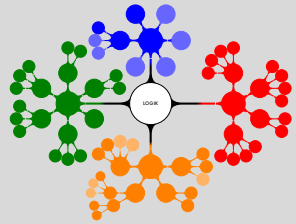
Programación manycore: GPUs

Sistemas Concurrentes y Distribuidos (SCD)- Doble Grado Ingeniería Informática Matemáticas

Asignatura *de GIIM (3er curso)*

Fecha 28 septiembre 2023

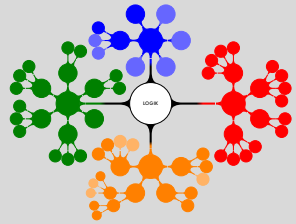
Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



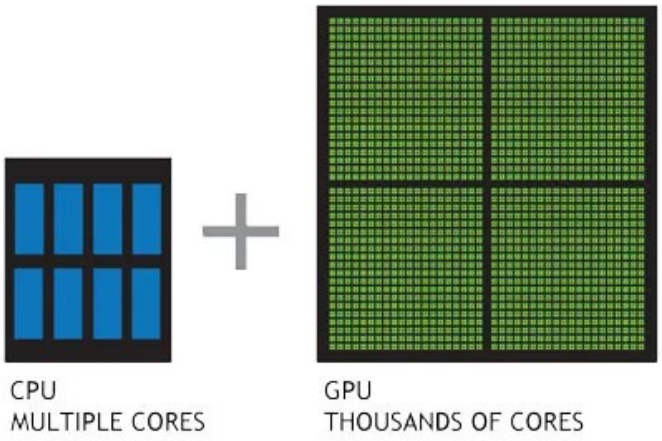
Arquitecturas heterogéneas de procesadores

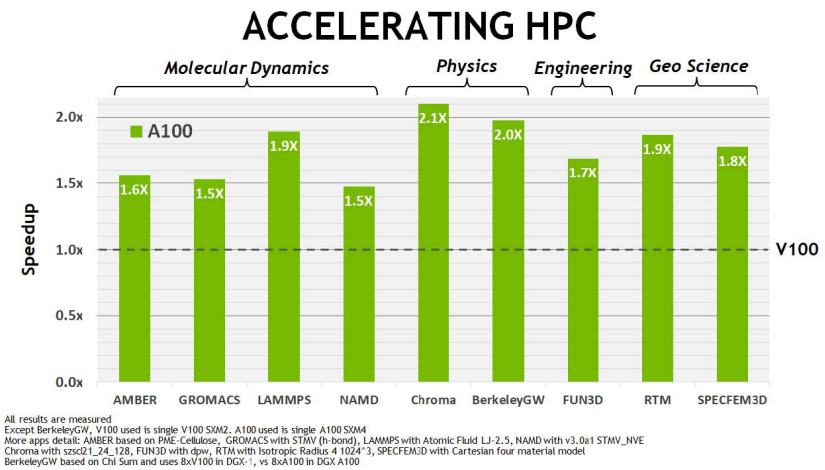
1. Sistemas ciber-físicos y Deep Learning poseen cada vez mayor carga computacional
2. Arquitecturas heterogéneas multinúcleo y aceleradores hardware:
   1. Graphical Processing Units (GPU)
   2. Neural Processing Units (NPU)
3. Procesamiento de datos masivos complejos que provienen de sensores de percepción avanzada
4. Los resultados de la computación no sólo han de ser correctos sino que se han de producir dentro de plazos de tiempo impuestos

Manuel I. Capel
manuelcapel@ugr.es

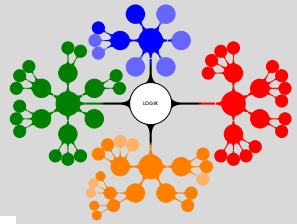


Comparativa de eficiencia CPU/GPU

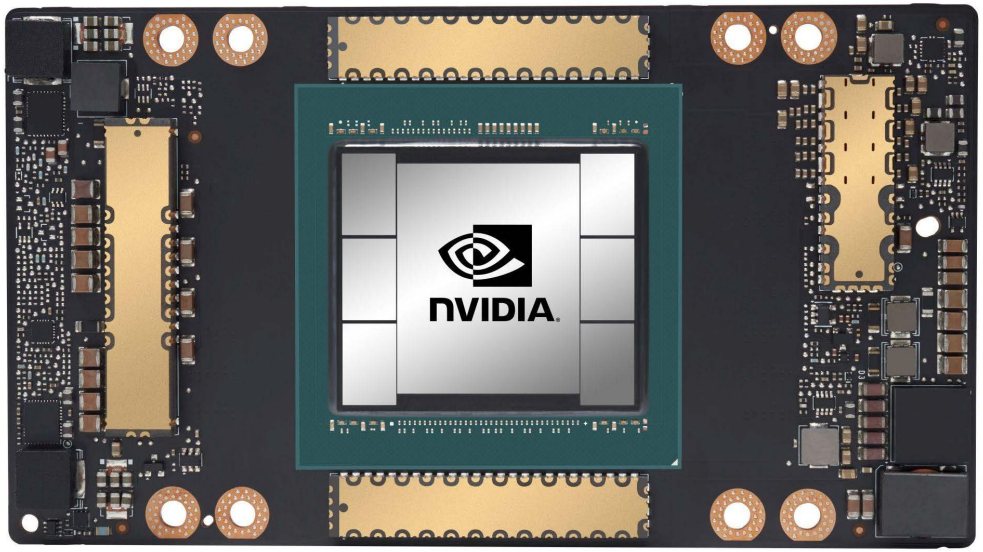




Manuel I. Capel
manuelcapel@ugr.es

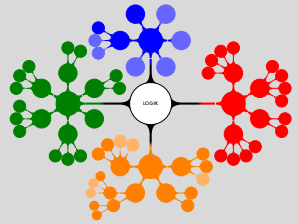


NVIDIA A100 GPU



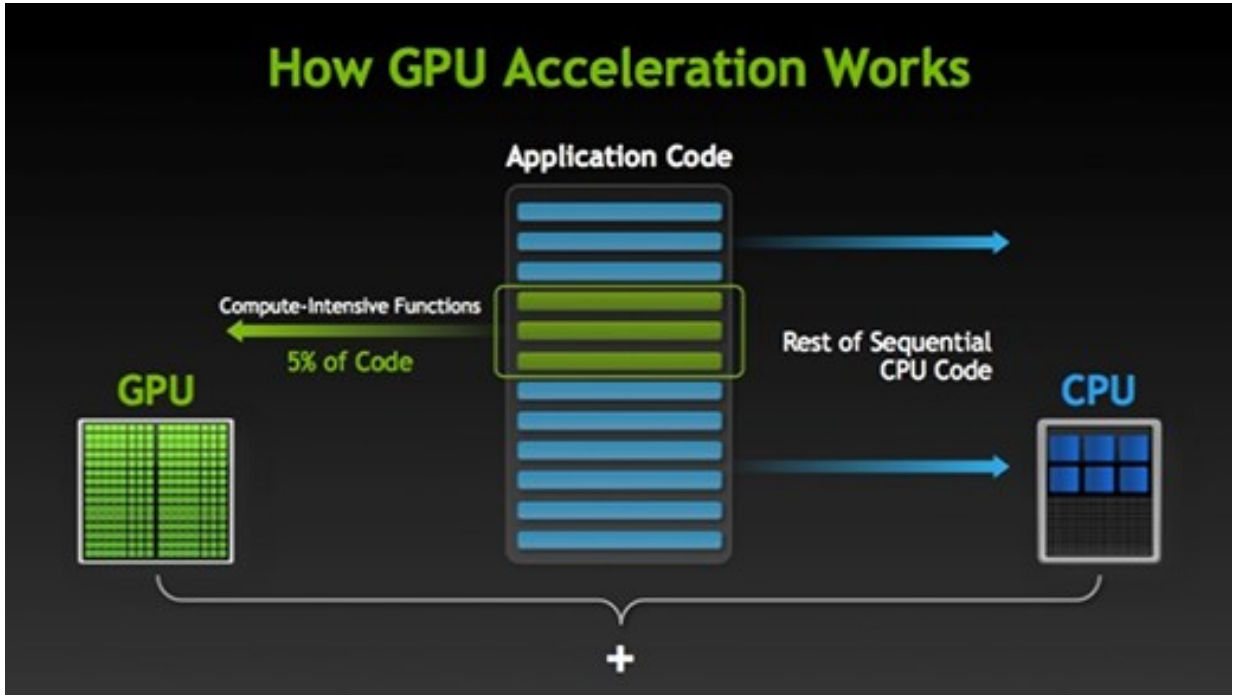
Las GPUs se integran en plataformas heterogéneas como co-procesadores junto a 1 ó más CPUs:
-Arquitecturas *discretas*: GPU en ranura de expansión
-Arquitecturas *integradas* (MPSoC): comunicación GPU-CPU a través de MC

Manuel I. Capel
manuelcapel@ugr.es



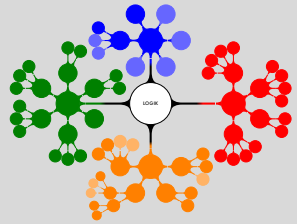
Funcionamiento básico de la plataforma heterogénea

- Información interna GPU no disponible
- 2 arquitecturas: NVDIA-CUDA y OpenCL



- Descarga de datos hacia el dispositivo acelerador
- Procesamiento en GPU
- Carga de datos de vuelta a la CPU

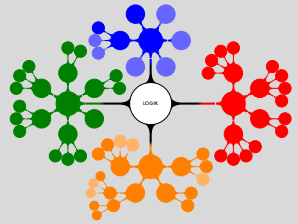
Manuel I. Capel
manuelcapel@ugr.es



Equivalencia entre terminología CUDA y OpenCL

CUDA	OpenCL
CUDA Core	Processing Elements
Warp	Wavefront
Streaming Multiprocessor (SM)	Compute Unit
(Thread) Block	Work-group
Thread	Work-item
Kernel	Kernel
Stream	Command Queue

Manuel I. Capel
manuelcapel@ugr.es



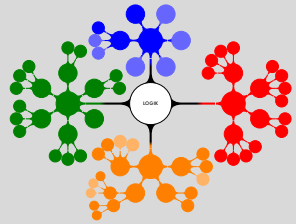
Arquitectura NVIDIA GPU

1 GPU se compone de:

- Unidades de procesamiento paralelo –Stream Multiprocessors (SM)
  - {Cores}
  - Registros
  - Niveles de cache
  - {Tensor cores}
- 1 core CUDA: ejecuta { Warps}
- Warp = { 32 Threads }

Modelo de procesameinto paralelo : Single Instruction Multiple Data (SIMD)
Se garantiza el “paralelismo de datos"
Warp Schedulers (puede haber más de 1): se encargan de ejecutar agrupaciones de cores CUDA en 1 mismo SM

Manuel I. Capel
manuelcapel@ugr.es

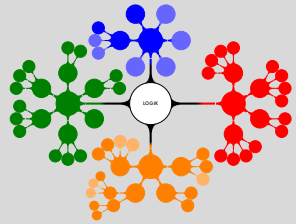


Ejecución de hebras por elementos de procesamiento en CUDA

De manera práctica:

- El programador escribe un “kernel", y organiza su ejecución en una malla (grid) de *thread-blocks*
- Cada bloque es asignado a 1 SM; una vez asignado, no puede migrar a otro SM
- Cada SM divide sus propios bloques (hasta un máximo de 32 por SM) en *Warps* y cada *Warp* tiene un tamaño máximo de 32 hebras
- Todas las hebras de un *Warp* se ejecutarán concurrentemente por los *cores* del SM
- Si un *Warp* contiene menos de 32 hilos, en la mayoría de los casos se ejecutará igual que si tuviera 32 hebras

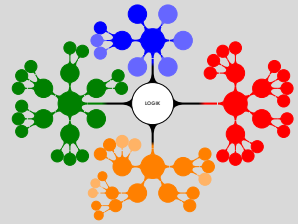
Manuel I. Capel
manuelcapel@ugr.es



Ejemplo de código CUDA-1: suma de 2 vectores grandes con 1 sola hebra

```
__global__ void add(int n, float *x, float *y)
{
        for (int i = 0; i < n; i++)
                y[i] = x[i] + y[i];
}
int main()
{
 int N = 1 << 20;
 float *x, *y;
 cudaMallocManaged(&x, N * sizeof(float));
 cudaMallocManaged(&y, N * sizeof(float));
 for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
 }
 add << <1, 1 >> > (N, x, y);
 cudaDeviceSynchronize();
 float maxError = 0.0f;
 for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i] - 3.0f));
 std::cout << "Max error: " << maxError << std::endl;
 cudaFree(x);
 cudaFree(y);
 return 0;
}
```


Manuel I. Capel
manuelcapel@ugr.es



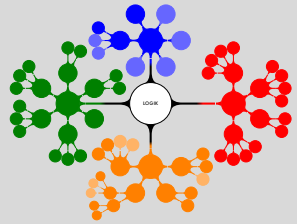
Asignación de `threads` a SMs

Cada bloque de hebras (*thread-block*) se divide en un número de *Warps* dado por la siguiente ecuación:

$$\texttt{WarpsPerBlock} = (\texttt{ThreadsPerBlock} + \texttt{WarpSize} - 1)/\texttt{WarpSize}$$

- Los *Warp Schedulers* del SM no respetan la contigüidad de los Warps en un mismo bloque cuando los asignan a los cores del SM
- Los *Warps* pueden bloquearse en las construcciones de barreras de los programas, en las operaciones de memoria, en las dependencias de datos, etc.
- Un *Warp* bloqueado no es eligible para ser seleccionado por el *Warp Scheduler* del elemento de procesamiento (SM)

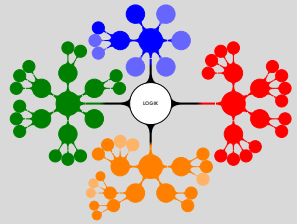
Manuel I. Capel
manuelcapel@ugr.es



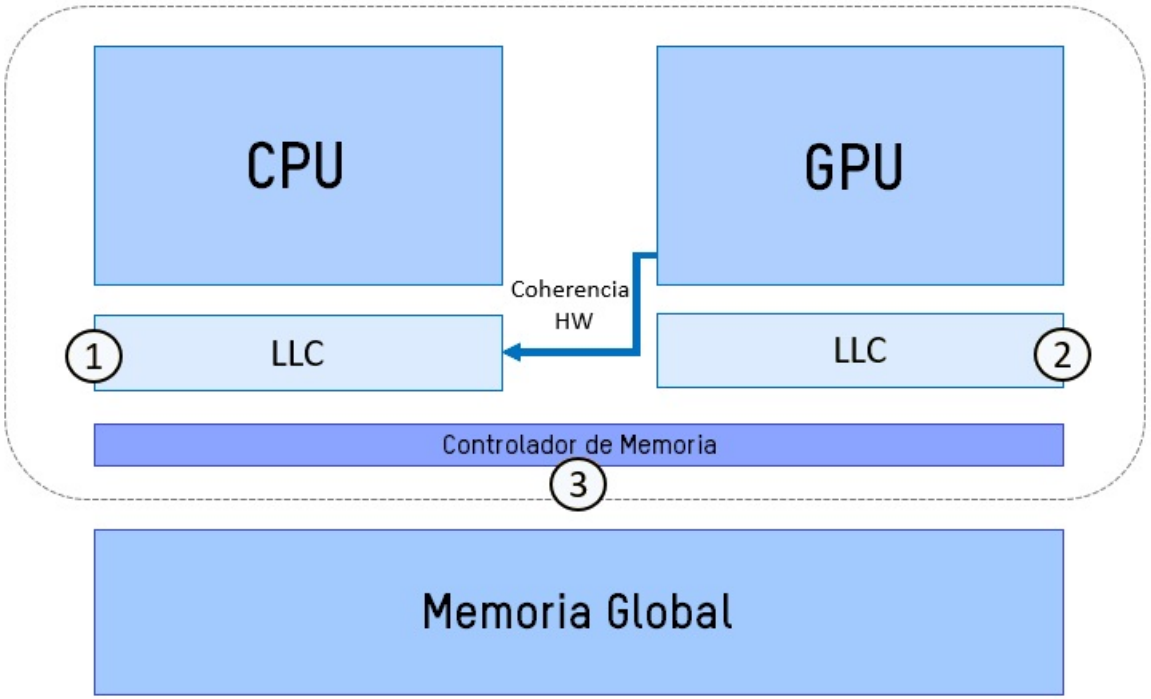
Ejemplo de código CUDA-1: suma de 2 vectores grandes con 4096 bloques y 256 hebras por bloque

```
__global__ void add(int n, float *x, float *y)
{ int index = blockIdx.x*blockDim.x + threadIdx.x;
  int paso = blockDim.x*gridDim.x;
  for (int i = index; i < n; i+= paso)
                y[i] = x[i] + y[i];
}
int main(){
 int N = 1 << 20;//1048576 elementos
 float *x, *y; int blockSize= 256;
 int numBlocks = (N + blockSize - 1) / blockSize;
 cudaMallocManaged(&x, N * sizeof(float));
 cudaMallocManaged(&y, N * sizeof(float));
 for (int i = 0; i < N; i++) {
  x[i] = 1.0f;  y[i] = 2.0f;
 }
 add << <numBlocks, blockSize >> > (N, x, y);
 cudaDeviceSynchronize();
 float maxError = 0.0f;
 for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i] - 3.0f));
 std::cout << "Max error: " << maxError << std::endl;
 cudaFree(x); cudaFree(y);
 return 0;
}
```


Manuel I. Capel
manuelcapel@ugr.es



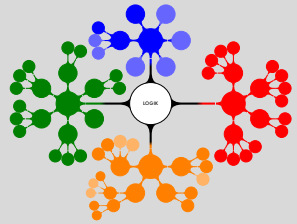
Organización general de la memoria en una GPU



Niveles de cache

-Reducir latencias provocadas por acceso a memoria global
-Las GPUs utilizan varios niveles de cache (dependiendo del modelo hardware)

Manuel I. Capel
manuelcapel@ugr.es

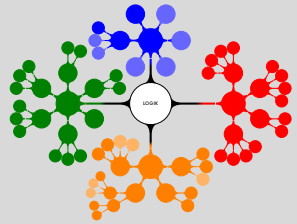


Modelo de ejecución de la GPU

Aspectos a considerar

1. A través de MC, la CPU carga un trozo de código ("kernel") escrito por el programador para que se ejecute en el acelerador (GPU)
2. Una tarea de la CPU lanza el "kernel":
   1. El *driver* del acelerador parte el "kernel" en varios bloques de hebras (*thread blocks*), que ejecutan las mismas instrucciones pero sobre diferentes porciones de datos
3. El *lanzamiento* de un "kernel" puede hacerse de manera síncrona o asíncrona
4. Existen varios mecanismos de programación para re-sincronizar los datos entre la CPU y la GPU
5. Los bloques de hebras se distribuyen en colas FIFO de operaciones en la GPU, llamadas *Streams*

Manuel I. Capel
manuelcapel@ugr.es



Modelo de ejecución de la GPU-II

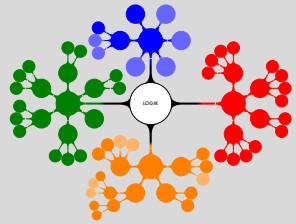
Operaciones en la GPU

1. Operaciones de copia de datos: procesadas por un *motor de copia*–Copy Engine (CE)
2. Operaciones de ejecución de los bloques de threads en los SM: procesadas por el motor de ejecución–Execution Engine (EE)
3. Los motores CE y EE son independientes y pueden funcionar concurrentemente
4. Se pueden realizar operaciones concurrentemente en la GPU asignando bloques de hebras a diferentes colas de ejecución con diferentes prioridades

Degeneración en ejecución secuencial

- El programador sólo ha asignado 1 Stream:
  1. Copia de datos de CPU a GPU
  2. Ejecución del “kernel"
  3. Copia de resultados de GPU a CPU

Manuel I. Capel
manuelcapel@ugr.es

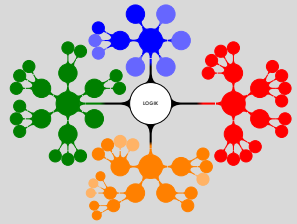


Modelo de ejecución de la GPU-III

Mecanismos de expulsión y arbitraje

- El programador puede provocar la expulsión de kernels asociados a diferentes *Streams*, pero sólo se produce cuando haya finalizado la ejecución de un bloque de hebras o una operación de copia
- Arbitraje del *Thread Block Scheduler*:
  - Utilizando “Ingeniería inversa” se ha conseguido asignar bloques de hebras de un “kernel” específico a un SM, observando como se asignan los bloques de hebras de las colas *Stream* a los SMs de una GPU
  - Existen 2 colas FIFO adicionales a las de los *Streams*: una para operaciones EE y otra para operaciones CE
  - La cola EE se encarga de encolar los bloques procedentes de un “kernel" antes de pasar a los SMs
  - La cola de CE contiene las operaciones de copia de datos a través del MC
  - Se pueden definir reglas de arbitraje para los bloques de copia y de “kernel" para moverse entre las colas anteriores

Manuel I. Capel
manuelcapel@ugr.es

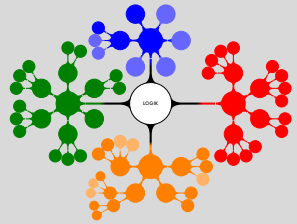


Modelo de ejecución de la GPU-IV

Ejecución dentro del SM

- Cuando 1 ó más bloques llegan al SM se descomponen en *Warps*
- Es posible ejecutar varios *warps* concurrentemente en un mismo SM o *core* CUDA dependiendo del número de “Warp Scheduler" que contenga la unidad de procesamiento
- Cada uno de estos planificadores se encargará de seleccionar un *Warp* que tenga 1 instrucción lista para ejecutarse y de darle acceso a los cores CUDA del SM
- Políticas de planificación de los “Warp Scheduler" :
  1. “Greedy-then-Older" (GTO) para arquitecturas de GPU discretas e integradas
  2. “Loose-Round-Robin" (RR), sobre todo se utiliza en arquitecturas integradas
  3. Con LRR los Warps se ejecutan de forma rotatoria hasta que 1 alcanza una dependencia insatisfecha, entonces se detiene la ejecución del Warp y pasa al siguiente (sin dependencia insatisfecha)

Manuel I. Capel
manuelcapel@ugr.es

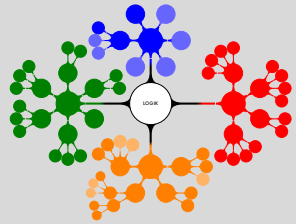


Comunicación a través de la memoria compartida (MC)

Planteamiento general de comunicación a través de la memoria

- La CPU comparte con la GPU la memoria global del sistema (en arquitecturas GPU integradas), como medio de comunicación entre ambos
- Suministrar a la GPU los datos que van a ser procesados en paralelo y posterior recopilación de los resultados de tal computación
- Modelos de comunicación utilizando CUDA:
  - Standard Copy (SC)
  - Zero Copy (ZC)
  - ZC + Coherencia hardware
  - Unifed Memory (UM)

Manuel I. Capel
manuelcapel@ugr.es

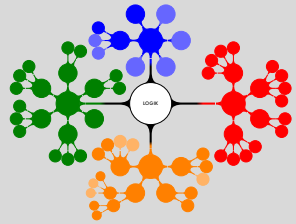


Modelos CUDA de comunicación entre CPU-GPU

Standard Copy(SC)

- En la memoria global del sistema se crean 2 espacios físicos: (1) CPU y (2) para GPU
- El mecanismo CE de la GPU transfiere datos sólo mediante acceso directo a memoria (DMA)
- Mecanismo muy simple y fácil de utilizar pero que puede presentar problemas de rendimiento debido a la latencia asociada con la transferencia de datos entre CPU y GPU
- La ventaja de usar SC sería en aplicaciones que permitan cargar todos los datos en la cache de la GPU y no tengan que acceder a la memoria global

Manuel I. Capel
manuelcapel@ugr.es

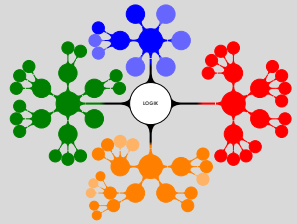


Modelos CUDA de comunicación entre CPU-GPU-II

Zero Copy(ZC)

- Utiliza un espacio común de la memoria compartida al que pueden acceder directamentetanto la CPU como la GPU
- La GPU puede acceder directamente a los datos sin necesidad de hacer transferencias o copias a memoria global
- Los datos se pasan a través de punteros en la GPU que apuntan a un espacio de direcciones compartido en la memoria global
- La cache de la GPU permanecerá inactiva
- El inconveniente de utilizar ZC es que tanto la CPU como la GPU tienen que acceder a los datos mediante punteros, anulando el efecto de los caches, lo que pueda dar lugar a tiempos de acceso más altos que con SC

Manuel I. Capel
manuelcapel@ugr.es

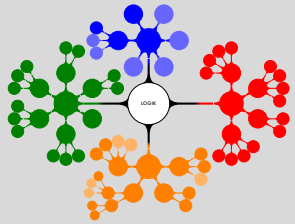


Modelos CUDA de comunicación entre CPU-GPU-III

Zero Copy(ZC)+Coherencia hardware

- Se optimiza el acceso ZC a memoria debido a que la GPU puede acceder al último nivel de cache de la CPU, evitando tener que acceder siempre a la memoria global
- ZC podría ser beneficiosa en comparación con SC, incluso mejorando los tiempos de ejecución del SC si este reduce los accesos a datos en la cache

Manuel I. Capel
manuelcapel@ugr.es

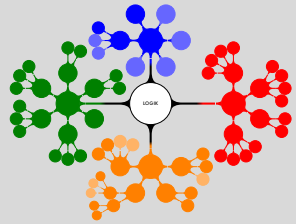


Modelos CUDA de comunicación entre CPU-GPU-III

Unified Memory (UM)

- La diferencia con SC es que la GPU y CPU acceden a una misma memoria física compartida aunque se mantienen espacios lógicos separados para los datos en el programa
- No es necesario realizar copias para compartir datos entre la CPU y la GPU
- El programador no tiene ningún control sobre el acceso físico a los datos, que queda bajo el control del driver
- Mejora el rendimiento de las aplicaciones porque minimiza el tiempo de latencia que sufre el mecanismo SC
- Se pierde predecibilidad de los tiempos de ejecución ya que el programador no tiene control
- UM suele funcionar bien para un tamaño limitado de memoria, a partir de una cierta cantidad se observa mayor lentitud que con SC
- El método `CUDA Alloc` sirve para asignar recursos adicionales en las aplicaciones y suele ser independiente de la plataforma y método de copia

Manuel I. Capel
manuelcapel@ugr.es

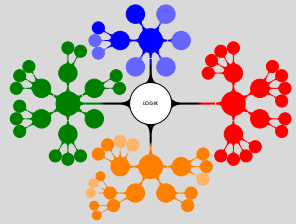


Origen de la interferencia entre CPU y GPU en la computación heterogénea

Interferencia por contención en el acceso a los diferentes niveles de memoria

- La interferencia por acceso intensivo a memoria por parte de la CPU y la GPU puede llegar a degradar la respuesta temporal de las tareas ejecutadas en ambas
- El primer punto de contención se encuentra en el último nivel de cache (LLC) de la CPU, que afecta a:
  - Los diferentes núcleos de la CPU puede que escriban concurrentemente en una línea de cache utilizada por los otros núcleos
  - Aumenta el tiempo de ejecución de las tareas y disminuye la predictibilidad de las aplicaciones
  - El mecanismo de coherencia entre CPU y GPU que supone utilizar el método ZC+coherencia hardware, ya que permite también a la GPU el acceso a la cache de la CPU

Manuel I. Capel
manuelcapel@ugr.es

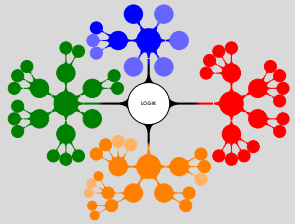


Origen de la interferencia entre CPU y GPU en la computación heterogénea-II

Interferencia por acceso al LLC de la CPU

- Interferencia entre kernels concurrentes y su impacto en el tiempo de ejecución de peor caso de los propios kernels
- En algunos casos, se obtiene menores tiempos de ejecución utilizando kernels concurrentes porque se aumenta la utilización global de la GPU

Manuel I. Capel
manuelcapel@ugr.es

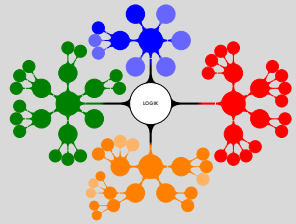


Origen de la interferencia entre CPU y GPU en la computación heterogénea-III

Interferencia por acceso a la memoria global

- Interferencia que se produce por saturación del controlador de memoria (MC)
- Al agotar el ancho de banda del MC se produce una latencia extra en las tareas quer intentan acceder a la memoria global
- En situación de saturación del MC, la ejecución de tareas en la CPU incrementará el peor tiempo de ejecución en la GPU y su variabilidad
- Además, la sincronización entre la CPU y la GPU es otra causa de interferencia, debido a puntos de sincronización programados en el código de las aplicaciones, tanto en el código que se ejecuta en la CPU como en la GPU
- Los bloqueos por sincronización implícita derivados de utilizar la API de CUDA

Manuel I. Capel
manuelcapel@ugr.es



Bibliografía

- NVIDIA A100 Tensor Core GPU Architecture Unprecedented Acceleration at Every Scale (accedido 2023)
- Yurtsever, E., Lambert, J., Carballo, A., Takeda, K., 2020. A survey of autonomous driving: Common practices and emerging technologies. IEEE Access 8, 58443?58469. doi:10.1109/ACCESS.2020.2983149
- Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D., 2018. Gpu scheduling on the nvidia tx2: Hidden details revealed. Real-Time Systems Symposium January, 104?115. doi:10.1109/RTSS.2017.00017
- Singh, J., Olmedo, I.S., Capodieci, N., Marongiu, A., Caccamo, M., 2022. Reconciling QoS and concurrency in nvidia gpus via warp-level scheduling. 2022 Design, Automation and Test in Europe Conference and Exhibition , 1275?1280doi:10.23919/DATE54114.2022.9774761
- Calderón, A.J., Torres, C., Kosmidis, L., Fernando, C., Ramírez, N., Javier, F.,Almeida, C., 2022. Real-Time High-Performance Computing for Embedded Control Systems. doi:10.5821/dissertation-2117-371621
- Capodieci, N., Burgio, P., Cavicchioli, R., Olmedo, I.S., Solieri, M., Bertogna, M., 2022. Real-time requirements for adas platforms featuring shared memory hierarchies. IEEE Design and Test 39, 35?41. doi:10.1109/MDAT.2020.3013828

Manuel I. Capel
manuelcapel@ugr.es