TPP – Tecnología de la Programación Paralela

Máster en Computación Paralela y Distribuida

2. Paradigma de Multi-Hilos Masivos en Aceleradores

José E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2015/16

Contenido

Apartado 1

Conceptos Básicos

- Arquitectura
- Kernels
- Hilos

Programación de Procesadores Gráficos (GPU)

Las GPUs proporcionan gran capacidad de proceso

- *Desktop supercomputing*
- Clusters híbridos

GPGPU: *General-purpose comput. on graphic processing units*

- Programación de bajo nivel
- Se basa en realizar una operación sencilla (*kernel*) sobre un conjunto de datos similares (*stream*)

Evolución actual: APIs/librerías/lenguajes/SDK

- Propietarios: CUDA (NVIDIA), Stream (AMD/ATI), DirectCompute (Microsoft)
- Multi-plataforma: OpenCL (estándar abierto definido por un comité), BrookGPU (académico)
- Directivas: OpenACC, OpenMP 4.0

Solución de NVIDIA

Hardware gráfico:

- GeForce: gama de consumo (juegos)
- Quadro: gama para estaciones de trabajo (CAD)
- Tesla: gama dedicada para computación

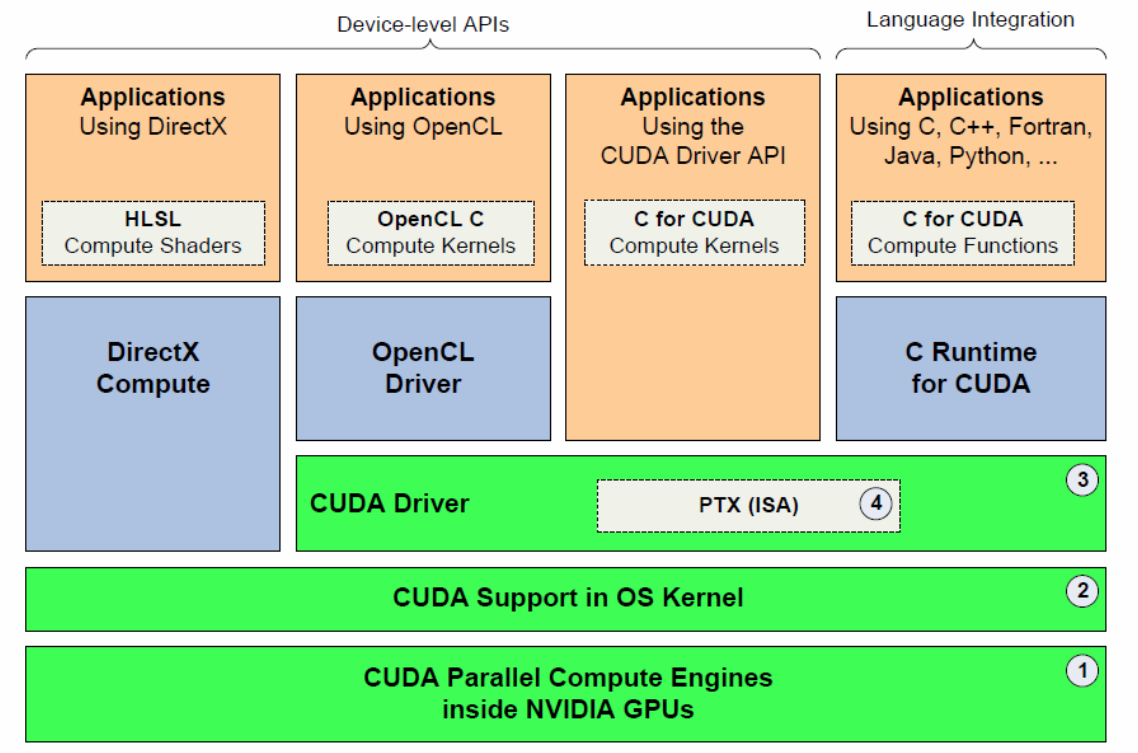
CUDA: *Compute Unified Device Architecture*

- Plataforma hardware-software
- CUDA C: extensión de C para programar GPUs
- Disponible desde 2007; versión actual: 7.5 (sep 2015)

Además del driver, incluye

- Compilador `nvcc` (C/C++ basado en LLVM)
- Herramientas: debug/profile, IDE, ...
- Librerías: cuBLAS, cuFFT, cuRAND, cuSPARSE, Thrust
- Ejemplos y documentación

Arquitectura CUDA



Interfaz de Programación

CUDA C: Extensión de C, permite definir *kernels* como funciones C

Normalmente se usa el *runtime* de CUDA (`cudart`)

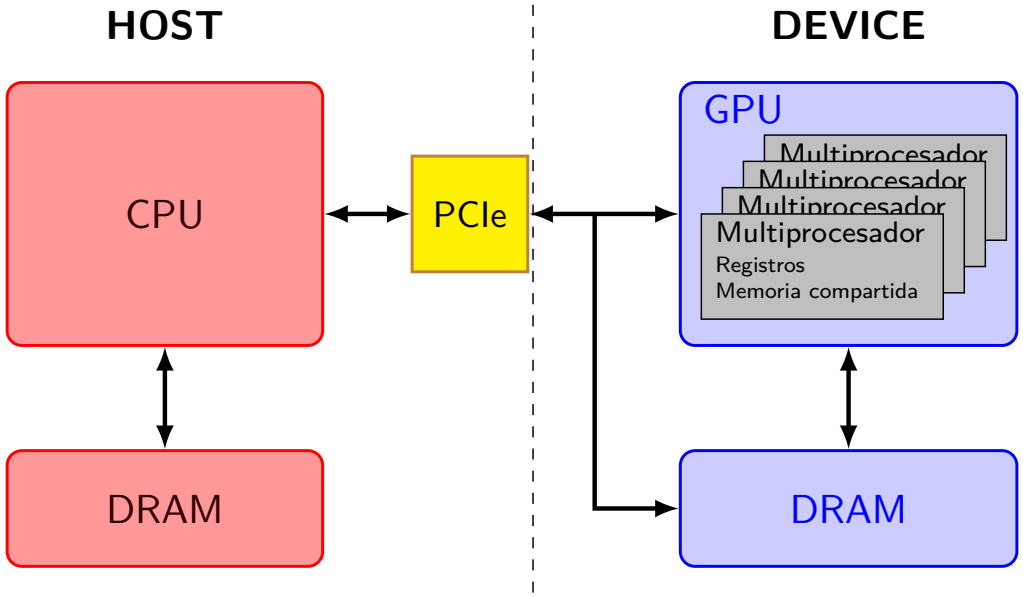
- Reservar y liberar memoria en el *device*
- Transferir datos entre memoria de *host* y *device*
- Gestionar múltiples *devices*

Ficheros `.cu` compilados con `nvcc`

- Compila el código de *host* con el compilador del sistema
- Para los *kernels* se genera PTX (pseudo-ensamblador)
- El *driver* genera código máquina específico para la GPU concreta (*JIT compiling*)

GPU como Co-procesador

CPU y GPU actúan como dispositivos separados, con sus propias DRAMs



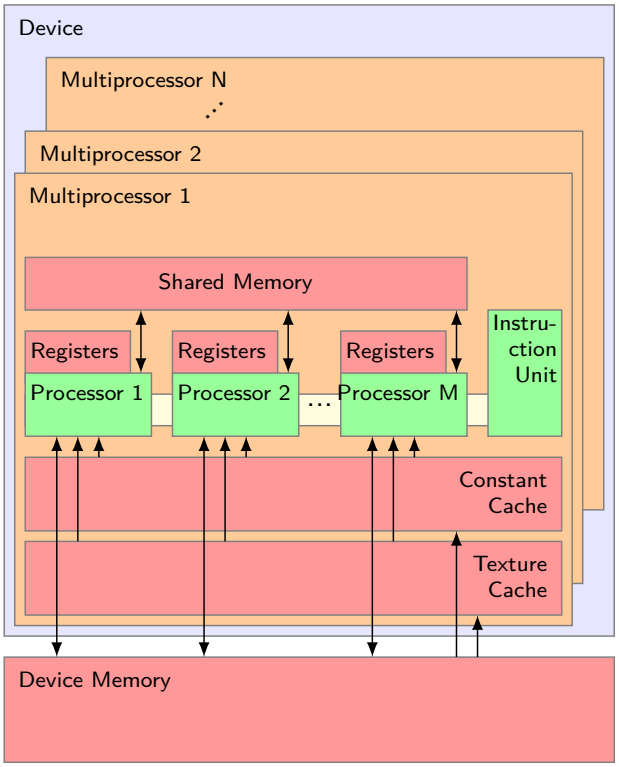
Modelo Hardware

N multiprocesadores

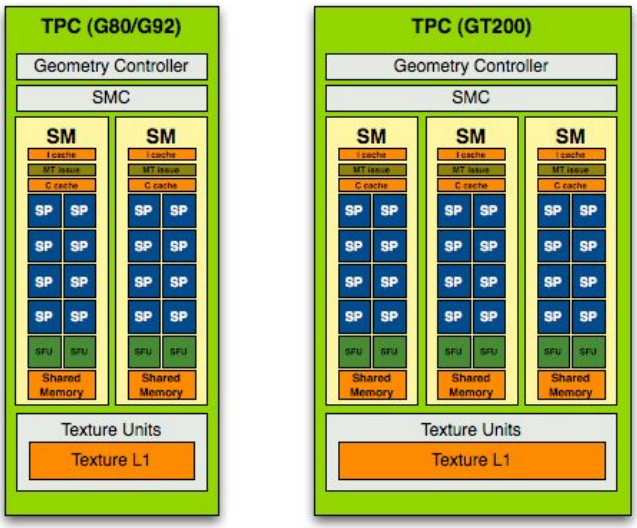
Cada uno incluye:

- M procesadores
- Banco de registros
- Memoria compartida: muy rápida, pequeña
- Memorias de constantes y texturas (solo lectura)

La memoria global es 500 veces más lenta que la memoria compartida



Micro-arquitectura Tesla



SM: *stream multiprocessor*
TPC: *texture/processor cluster*
SP: *streaming processor*

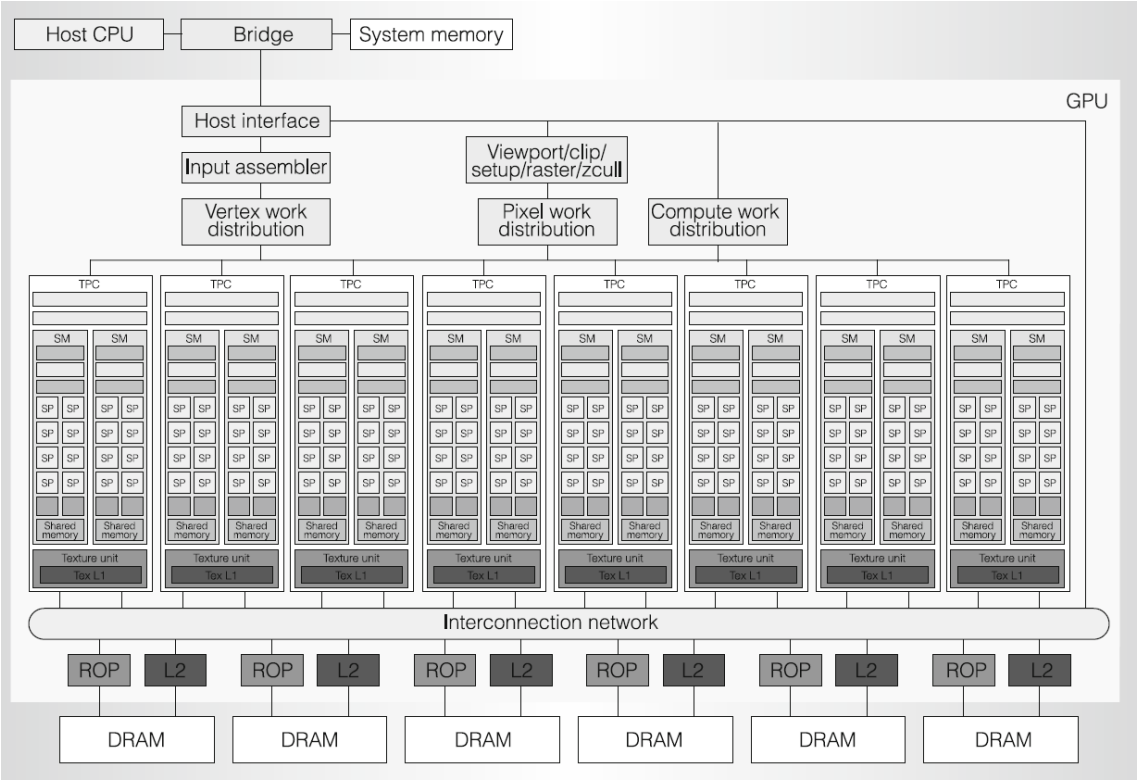
Serie 8 (G80)

- 128 núcleos
- SM con 8 núcleos, shared 16 KB, constant 8 KB

Serie 200 (GT200)

- 240 núcleos
- SM con 8 núcleos, memoria 16 KB/8 KB, 1 unidad de doble precisión

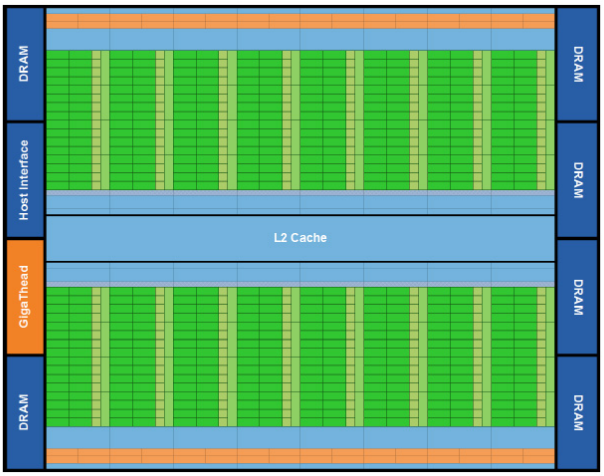
Ejemplo: G80/G92

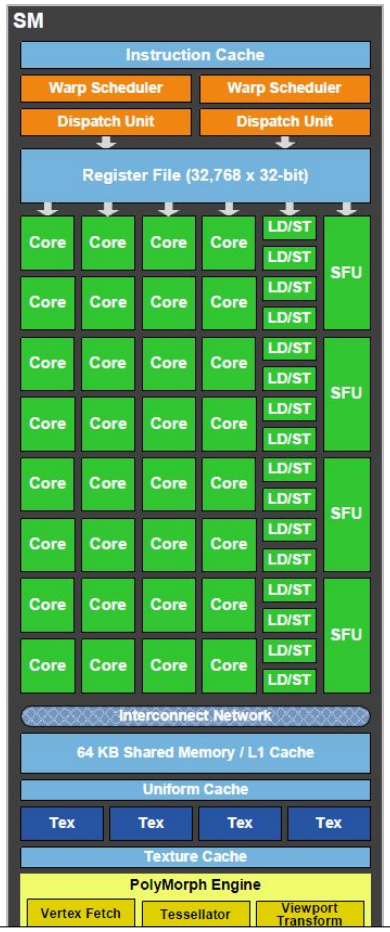


Micro-arquitectura Fermi

512 nucleos, 16 SM con:

- 32 núcleos
- 16 unidades de doble precisión
- 64 KB de SRAM a repartir entre memoria compartida y cache L1
- Cache L2 común

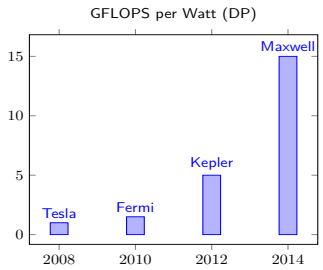




Micro-arquitecturas Nuevas

Kepler (2012)

- 192 núcleos en cada SM
- Hasta 2688 núcleos en total (TITAN)
- 6 GB de memoria



Maxwell (2014)

- Énfasis en mejorar eficiencia energética
- Misma potencia con menos núcleos

Pascal (2016?)

- Memoria 3D, ancho de banda 1 TB/s
- Memoria unificada CPU-GPU
- Bus de alta velocidad (un orden de magnitud mayor que PCI Express)

CUDA Compute Capability

Compute Capability es como una versión del hardware

- El número mayor indica la micro-arquitectura: 1=Tesla, 2=Fermi, 3=Kepler, 4=Maxwell, ...
- El número menor se va incrementando a medida que se van añadiendo nuevas prestaciones en cada arquitectura

Ejemplo: paralelismo dinámico disponible en CC 3.5+

Al compilar se puede indicar una *compute capability* específica:

```
$ nvcc -arch=sm_21 -o mytest mytest.cu
```

Ejemplo `deviceQuery`

```
Device 0: "GeForce GTX 280"
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.20
  CUDA Capability Major/Minor version number:    1.3
  Total amount of global memory:                 1073020928 bytes
  Multiprocessors x Cores/MP = Cores:            30 x 8 = 240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default
  Concurrent kernel execution:                   No
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
```


Kernels CUDA

Kernel: Función que se ejecuta N veces por N hilos diferentes

- Se declaran con el modificador `__global__`
- ~~No se puede ejecutar varios *kernels* a la vez en un *device*~~

Restricciones:

- No puede acceder a memoria del *host*
- Nº argumentos no variable
- Devuelve tipo `void`
- No recursivo
- Sin variables `static`

Suma de vectores

```
__global__ void VecAdd (float* A,
                        float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main()
{
  ...
  // Invocacion del kernel
  VecAdd<<<1,N>>>(A,B,C);
}
```

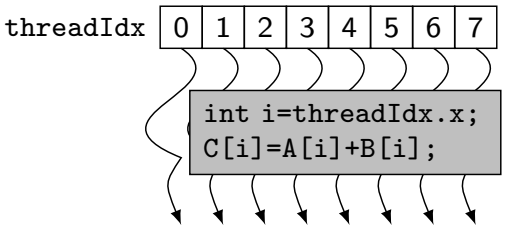
Hilos CUDA

Extremadamente ligeros

- Creación y cambio de contexto rapidísimos
- Para lograr eficiencia: descomposiciones de grano fino que permitan lanzar miles de hilos

Un kernel lo ejecutan un conjunto de hilos:

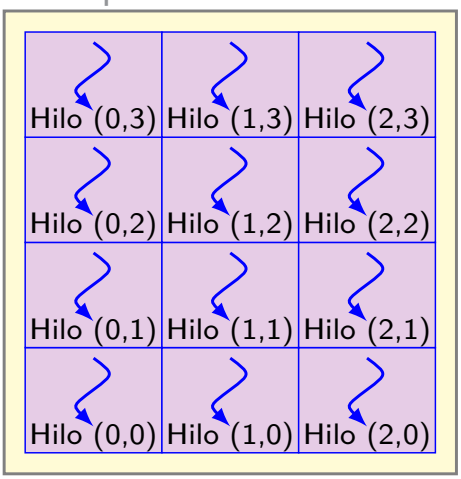
- Todos ejecutan el mismo código pero la acción depende del id. de hilo
- El id. de hilo se usa para calcular direcciones de memoria y tomar decisiones de control



Bloques de Hilos

Cada hilo se identifica con un índice (`threadIdx`) unidimensional, bidimensional o tridimensional dentro de un bloque de hilos

- Unidad de asignación de hilos a multiprocesadores
- El número de hilos por bloque está limitado
- Los hilos de un bloque pueden comunicarse a través de memoria compartida
- La variable predefinida `blockDim` contiene las dimensiones del bloque



`blockDim.x`=3, `blockDim.y`=4

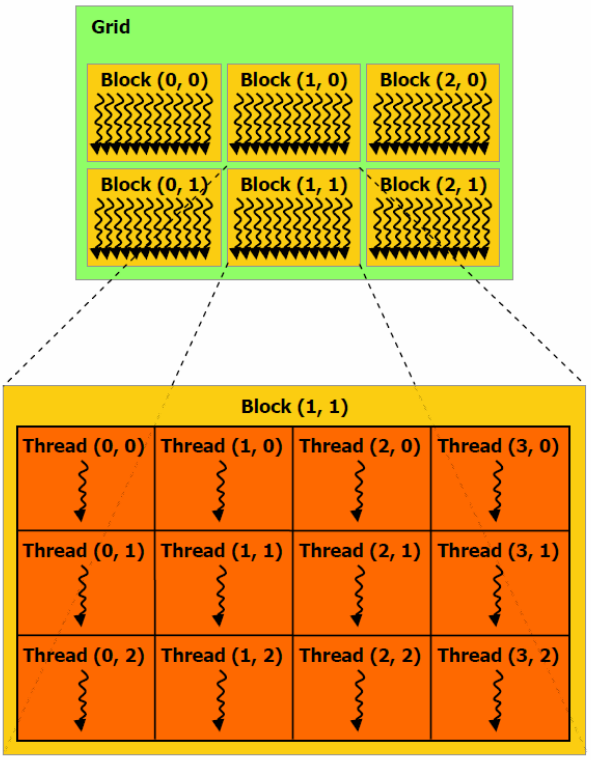
Grid de Bloques

Los bloques se agrupan en un *grid*

El número de hilos por bloque y bloques por *grid* se especifican al lanzar el *kernel*

Variables predefinidas

- `uint3 blockIdx`: índice del bloque dentro del *grid*
- `dim3 gridDim`: dimensiones del *grid*



SIMT vs SIMD

SIMT: *single instruction multiple threads*

- Todos los núcleos ejecutan la misma instrucción simultáneamente sobre distintos datos

Clave para alto rendimiento: muchos hilos

- Cada hilo tiene sus propios registros (esto limita el número máximo de hilos activos)
- WARP: grupo de 32 hilos que se ejecutan a la vez
- La ejecución alterna entre warps activos e inactivos (para enmascarar latencia de acceso a memoria)
- Los warps se separan en 2 half-warps de 16 hilos
- En ocasiones (p.e. un salto) los hilos *divergen* y deben ejecutarse secuencialmente $\rightarrow$ pérdida de prestaciones, pero gran flexibilidad en comparación con SIMD

Apartado 2

Programación

- Kernels
- Gestión de Memoria
- Memoria Compartida

Kernel de Ejemplo

Suma de matrices $N \times N$ - Kernel

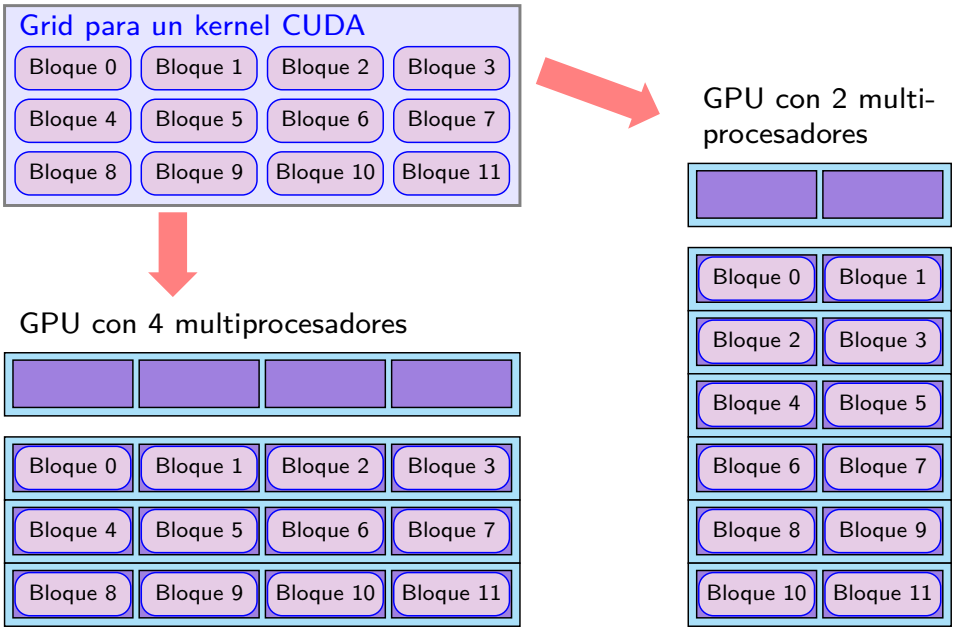
```
__global__ void MatAdd (float A[N][N], float B[N][N],float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i<N && j<N) C[i][j] = A[i][j] + B[i][j];
}
```

Suma de matrices $N \times N$ - Código del *host*

```
int main()
{
  ...
  // Invocacion del Kernel
  dim3 threadsPerBlock(16,16);
  dim3 numBlocks(N/threadsPerBlock.x,N/threadsPerBlock.y);
  MatAdd <<<numBlocks, threadsPerBlock>>> (A, B, C);
}
```

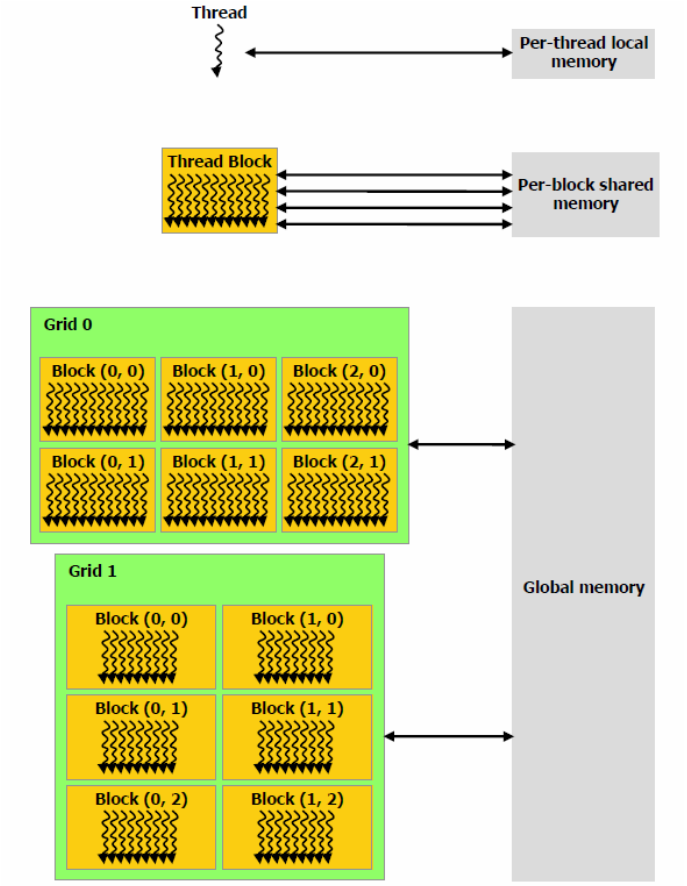
Escalabilidad Automática

El hardware se encarga de asignar los bloques a multiproc.: los *kernels* escalan y se adaptan al número de núcleos disponibles



Jerarquía de Memoria

- Los hilos pueden acceder a registros y memoria local individualmente
- La memoria compartida permite intercambios a nivel de bloque
- La memoria global, de constantes y texturas es persistente entre llamadas a *kernels*

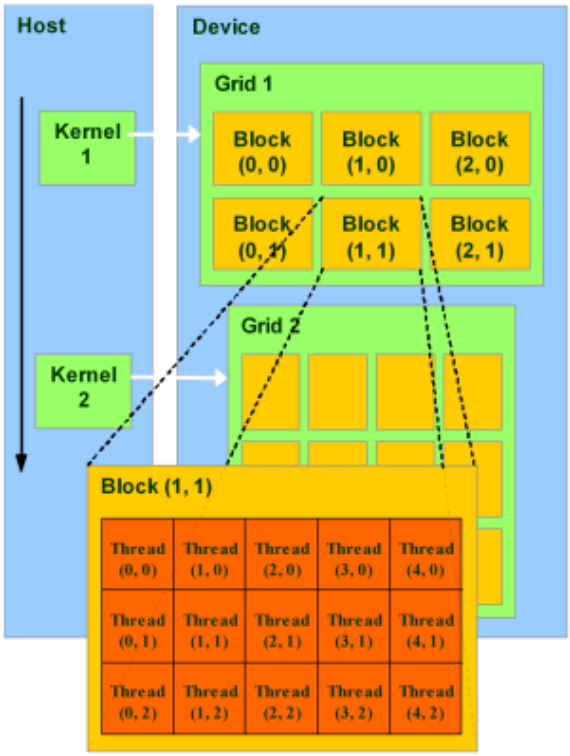


Programación Heterogénea

La ejecución de los *kernels* se intercala con la ejecución secuencial en la CPU

Un kernel no comienza en GPU hasta que no hayan finalizado las llamadas CUDA anteriores

`cudaThreadSynchronize()` espera a que se completen las operaciones previas



Modificadores

Modificadores para funciones a ejecutar en GPU

- Función invocada desde CPU:
  `__global__ void mikernel(...) {.....}`
- Función invocada desde GPU:
  `__device__ void mifuncion(...) {.....}`

Modificadores para variables en *device*

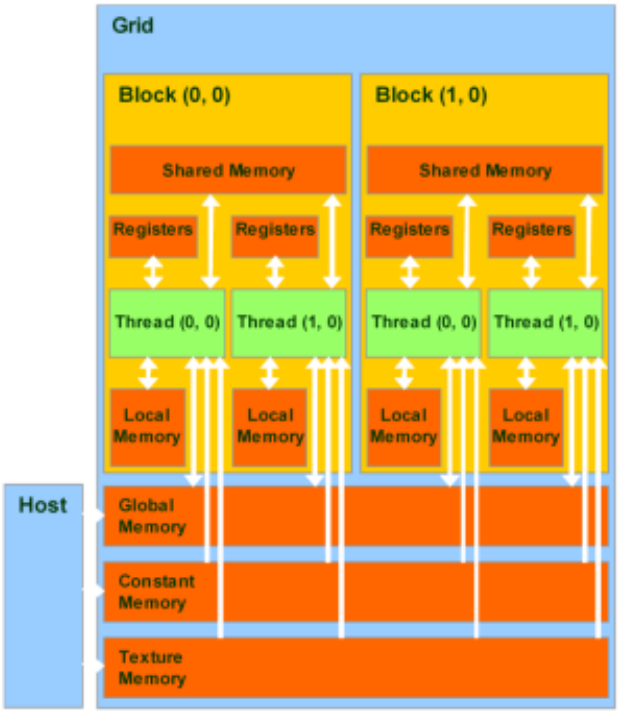
- Variable en memoria compartida:
  `__shared__ float matriz[32];`
  - Accesible por todos los hilos dentro del mismo bloque
  - Solo dura mientras se ejecuta el bloque de hilos
- Constante: `__constant__ float A[64];`
- También `__local__` y `__device__` (global)
- Variables automáticas sin modificador: se almacenan en registros si caben; si no, en memoria local

Gestión de Memoria

CPU y GPU tienen espacios de memoria separados

En el *runtime* hay funciones para:

- Reservar/liberar memoria global del *device*
- Transferir datos entre memoria global *device* y memoria *host*



Reservar/Liberar Memoria

- Reservar memoria en *device*:
  `cudaMalloc(void **ptr, size_t numbytes)`
- Liberar memoria en *device*:
  `cudaFree(void* ptr)`
- Asignar valor:
  `cudaMemset(void *ptr, int valor, size_t numbytes)`

Ejemplo de uso de memoria

```
int numbytes = 1024*sizeof(int);
int *d_x;
cudaMalloc( (void**)&d_x, numbytes );
cudaMemset( d_x, 0, numbytes);
...
cudaFree(d_x);
```

Se recomienda distinguir variables de *host* y *device* con `h_*` y `d_*`, respectivamente

Transferencias de Datos

```
cudaMemcpy(void *destino, void *fuente, size_t nbytes,
           enum cudaMemcpyKind direccion)
```

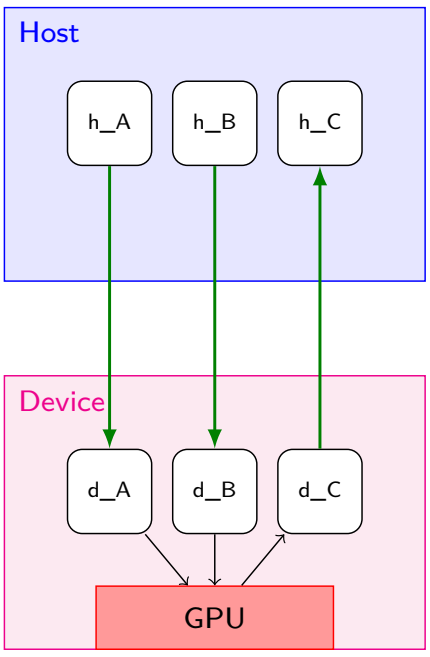
- La localización (*host* o *device*) de `destino` y `fuente` vienen dados por `direccion`:
  - `cudaMemcpyHostToDevice`: desde la CPU a la GPU
  - `cudaMemcpyDeviceToHost`: desde la GPU a la CPU
  - `cudaMemcpyDeviceToDevice`: entre posiciones de la memoria global *device*
- La llamada bloquea al hilo CPU y devuelve el control cuando se completa la copia de datos
- La transferencia no se inicia hasta que se hayan completado todas las llamadas CUDA previas

Transferencias de Datos - Ejemplo

```
__global__
void VecAdd(float* A,float* B,float* C,int N) {
  int i = blockDim.x*blockIdx.x + threadIdx.x;
  if (i<N) C[i] = A[i] + B[i];
}

int main()
{
  int N = ...;
  int nhilos = 256;
  int nbloques = ceil(float(N)/nhilos);
  size_t size = N*sizeof(float);
  float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
  h_A = (float*) malloc(size);
  h_B = (float*) malloc(size);
  h_C = (float*) malloc(size);
  initialize(h_A, h_B, N);
  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  VecAdd<<<nbloques,nhilos>>>(d_A, d_B, d_C, N);
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);

}
```



Uso de la Memoria Compartida

La memoria compartida es mucho más rápida que la global

- Si se puede, interesa reemplazar accesos a memoria global por accesos a memoria compartida
- Puede suponer rediseñar el código para reutilizar datos en memoria compartida

---

Sincronización de Hilos

```
void __syncthreads()
```

- Barrera entre todos los hilos del bloque
- Para evitar inconsistencias en acceso a mem. compartida

Solo se puede llamar en código condicional si la condición se evalúa igual en todos los hilos del bloque

Producto Matriz-Vector

```
#define num_threads 64
#define bs num_threads

__global__
void matvec_kernel(int n,double *A,int lda,double *x,double *y)
{
  int i,j,ind = blockIdx.x*num_threads + threadIdx.x;
  __shared__ double buff[bs];
  double res = 0.0;

  A += ind;
  x += threadIdx.x;
  for (i=0; i<n; i+=bs) {
    __syncthreads();
    buff[threadIdx.x] = x[i];
    __syncthreads();
    for (j=0; j<bs; j++) {
       res += A[0]*buff[j];
       A += lda;
    }
  }
  y[ind] = res;
}
```

Producto Matriz-Matriz (1)

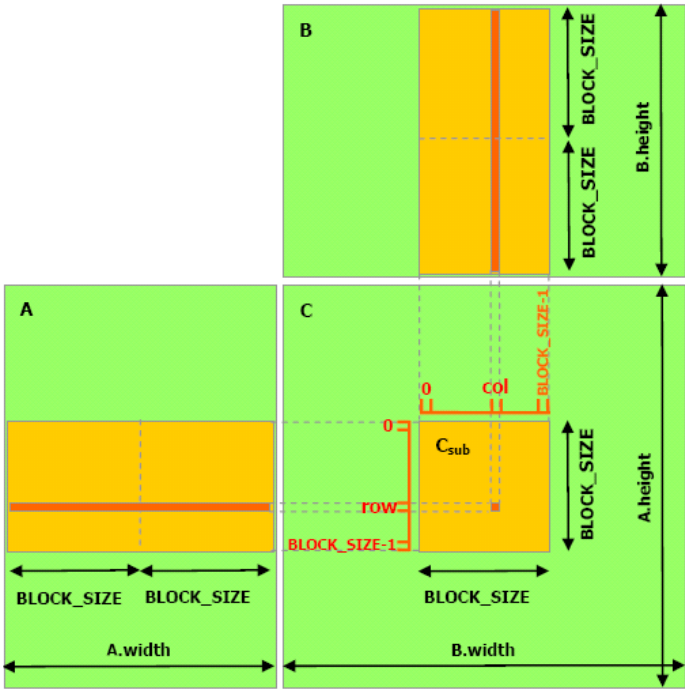
Cada bloque calcula una submatriz C_{sub}, un elemento por hilo

```
// reserva memoria en host
int size_A = sizeof(float)*nA*mA;
int size_B = sizeof(float)*nB*mB;
int size_C = sizeof(float)*nC*mC;
float* h_A = (float*)malloc(size_A);
float* h_B = (float*)malloc(size_B);
float* h_C = (float*)malloc(size_C);

// reserva memoria y copia matrices
cudaMalloc((void**) &d_A, size_A);
cudaMalloc((void**) &d_B, size_B);
cudaMalloc((void**) &d_C, size_C);
cudaMemcpy(d_A, h_A, size_A,
          cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size_B,
          cudaMemcpyHostToDevice);

// llamada al kernel
dim3 thrds(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(nC/thrds.x, mC/thrds.y);
matrixMul<<<grid,thrds>>>(d_C, d_A,
          d_B, nA, nB);

// copia resultado al host
cudaMemcpy(h_C, d_C, size_C,
          cudaMemcpyDeviceToHost);
```



Producto Matriz-Matriz (2)

```
__global__ void matrixMul( float* C, float* A, float* B, int nA, int nB)
{
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;

   int aBegin = nA*BLOCK_SIZE*by;     // Indice de primera submatriz
   int bBegin = BLOCK_SIZE*bx;

   float Csub = 0;    // Elemento calculado por el hilo

   // Recorrer todas las submatrices de A y B
   for (int a = aBegin, b = bBegin;
         a <= aBegin + nA - 1;
         a += BLOCK_SIZE, b += BLOCK_SIZE*nB) {

      // Copia submatrices de A y B en mem. compartida; cada hilo carga un solo elemento
      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
      As[ty][tx] = A[a + nA * ty + tx];
      Bs[ty][tx] = B[b + nB * ty + tx];

      __syncthreads();       // Sincroniza para asegurar carga completa

      // Multiplica las matrices, cada hilo calcula un elemento
      for (int k = 0; k < BLOCK_SIZE; k++)
         Csub += As[ty][k] * Bs[k][tx];

      __syncthreads();       // Sincroniza para asegurar fin de calculo
   }

   // Almacena sub-matriz calculada, cada hilo copia un elemento
   int c = nB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
   C[c + nB * ty + tx] = Csub;
}
```

Apartado 3

Aspectos Avanzados

Acceso a Memoria Coordinado: *Coalescing*

El acceso a memoria global puede ser más o menos eficiente

Coalescing es un acceso coordinado por un *warp* o *half-warp*

- Acceso a región contigua (64, 128 o 256 bytes)
- Dirección de inicio ha de ser múltiplo del tamaño de la región (*alignment*)
- El hilo k del *warp* accede al elemento k de los bloques
- Se traen los datos de todos los hilos en una sola carga
- No es necesario que participen todos los hilos, ni que lo hagan en orden (en las primeras arquitecturas las condiciones eran muy estrictas, luego se han relajado)

Hay que tener en cuenta que en 2D el hilo (x, y) ocupa la posición $x + n_x \cdot y$, y en 3D $x + n_x \cdot (y + n_y \cdot z)$

Errores y Depuración

Todas las llamadas a CUDA (excepto al lanzar un *kernel*) retornan un código de error (`cudaError_t`)

- `cudaSuccess`: ausencia de error
- `cudaGetErrorString`: para imprimir mensaje
- `cudaGetLastError`: código de error de la última llamada

Para depurar se pueden incluir en el *kernel* llamadas a `printf`

- La salida se vuelca a un espacio en memoria global (limitado)
- Tras la ejecución se copia al *host* y se saca por consola

Para depuración más avanzada usar herramientas: `cuda-memchk` y `cuda-gdb`

Sincronización Avanzada

La primitiva `__syncthreads()` tiene variantes para, además de sincronizar, evaluar una condición en todos los hilos de un bloque

Sincronización con evaluación

```
int __syncthreads_count(int predicate)
int __syncthreads_and(int predicate)
int __syncthreads_or(int predicate)
```

- La variante `count` devuelve el número de hilos en que se cumple la condición
- La variante `and` devuelve no-cero si se cumple en todos
- La variante `or` devuelve no-cero si se cumple en alguno

Tipos de Memoria

Tipo	Ubicación	Cache	T. Acceso
Registros	Multiprocesador	Sí	1 ciclo
M. compartida	Multiprocesador	Sí	1 ciclo
M. constante	DRAM de la tarjeta	Sí	1–100 ciclos
M. texturas	DRAM de la tarjeta	Sí	100 ciclos
M. superficies	DRAM de la tarjeta	Sí	100 ciclos
M. global	DRAM de la tarjeta	No	Muy lenta

- Los registros limitan el número de hilos
- La optimización estándar consiste en usar memoria compartida siempre que sea posible; si no, memoria constante para datos pequeños y de textura para más grandes

La memoria *shared* se puede reconfigurar como cache L1
`cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)`

Tipos de Arrays en la GPU

La forma básica es usar `cudaMalloc` y `cudaMemcpy` para reservar memoria y transferir datos

Existen formas específicas de gestionar la memoria:

- `cudaArray`: estructura especial para usar en memoria de textura; se gestiona con `cudaMallocArray`, `cudaMemcpyToArray`, ...
- `cudaMallocPitch` y `cudaMemcpy2D`: para datos bidimensionales
- `cudaMalloc3D` y `cudaMemcpy3D`: para datos tridimensionales

Estas primitivas fuerzan el alineamiento de memoria para cada parte (*padding*)

Tipos de Arrays en Host

En el host, se puede reservar memoria de dos tipos:

- Normal (paginada), con el `malloc` del sistema
- Bloqueada (*pinned* o *page-locked*)
  - Mediante `cudaHostAlloc` y `cudaHostFree`
  - `cudaHostRegister` permite bloquear un array normal
  - Ventaja: más eficiente, se evita una copia
  - Ventaja: la copia se puede solapar con la ejecución de un *kernel*
  - Ventaja: en arquitecturas nuevas se puede mapear directamente a direcciones de *device* (sin copia explícita)

Otros tipos: *portable*, *write-combining*, *mapped*

Medida de Tiempos: Eventos

Para tomar tiempos, se pueden usar funciones del host como `gettimeofday()` llamando a `cudaThreadSynchronize()`

Sin embargo, los eventos de CUDA tienen mejor resolución

CUDA Events

```
/* crear eventos */
cudaEvent_t start, finish;
cudaEventCreate(&start);
cudaEventCreate(&finish);
/* registrar eventos antes y después */
cudaEventRecord(start,0);  /* 0 es el stream por defecto */
my_kernel<<<dimGrid,dimBlock>>>(...);
cudaEventRecord(finish,0);
/* sincronizar */
cudaEventSynchronize(start);  /* opcional */
cudaEventSynchronize(finish); /* espera a que termine el evento */
/* calcular diferencia */
float elapsedTime;
cudaEventElapsedTime(&elapsedTime,start,finish);
```

Operaciones Atómicas

Las funciones atómicas realizan operaciones de lectura-modificación-escritura sobre una variable de 32 o 64 bits en memoria global o compartida

- Se garantiza que cada hilo la ejecuta sin interrupción
- Un uso típico es en las reducciones

Reducción de suma básica

```
__global__
void sum_reduce1(int *vector, int *sum, int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if (idx<N) atomicAdd(sum,vector[idx]);
}

sum_reduce1<<<numBlocks,threadsPerBlock>>>(d_x, &sum, N);
```


Operaciones Atómicas

Reducción de suma optimizada

```
__global__
void sum_reduce2(int *vector, int *sum, int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  __shared__ int sdata[1];
  if (threadIdx.x==0) {
    sdata[0]=0;
  }
  __syncthreads();
  if (idx<N) atomicAdd(&sdata[0],vector[idx]);
  __syncthreads();

  if (threadIdx.x==0) {
    atomicAdd(sum,sdata[0]);
  }
}
```

Reducción *Butterfly*

Reducción sin operaciones atómicas, con más sincronizaciones

```
__global__
void sum_reduce3(int* vector, int *sum, int N)
{
  // B es una potencia de 2 (constante)
  int i = threadIdx.x;
  __shared__ int psum[B];

  psum[i] = vector[i];
  __syncthreads();
  for (int bit = B/2; bit > 0; bit /= 2) {
    int inbr = (i + bit) % B;
    int t = psum[i] + psum[inbr];
    __syncthreads();
    psum[i] = t;
    __syncthreads();
  }
  *sum = psum[0];
}

sum_reduce1<<<1,N>>>(d_x, &sum, N);
```


Copia Asíncrona y Ejecución Concurrente

El modo avanzado para lanzar *kernels* se basa en el concepto de *stream*

- Por defecto hay un solo *stream*
- Para usar varios: `cudaStreamCreate`, `cudaStreamDestroy`

La copia asíncrona entre *host* y *device* requiere un *stream*

```
cudaMemcpyAsync(d_x,h_x,nbytes,cudaMemcpyHostToDevice,stream[0])
```

La ejecución de *kernels* tiene un argumento opcional para indicar el *stream*

```
myKernel_0<<<dimGrid,dimBlock,0,stream[0]>>>(d_x,N)
```

- Permite solapar varios *kernels* y operaciones de memoria
- También el uso de varias GPUs, con `cudaSetDevice`
- Hay primitivas para sincronizar a nivel de *stream*

Apartado 4

OpenCL

- Conceptos Básicos
- Ejecución de Programas

Introducción a OpenCL

OpenCL es un entorno de desarrollo multi-plataforma

- Desarrollado inicialmente por Apple (1.0 en Snow Leopard, 2008), más tarde por Khronos
- Versión estable: 2.1 (nov 2015)

Ventajas principales:

- Cubre más tipos de dispositivos, incluyendo Xeon Phi
- Se puede ejecutar código OpenCL en CPU y GPU

En OpenCL el código se organiza de forma distinta a CUDA

- En CUDA el compilador genera código a partir de los ficheros `*.cu`
- En OpenCL los ficheros `*.cl` se compilan en la ejecución
- Esto implica que hay que crear una máquina virtual que cargue y compile los ficheros `*.cl`

Modelo de Ejecución y Memoria

Los *kernels* se ejecutan por al menos un *work-item*, y los *work-items* se organizan en *work-groups*

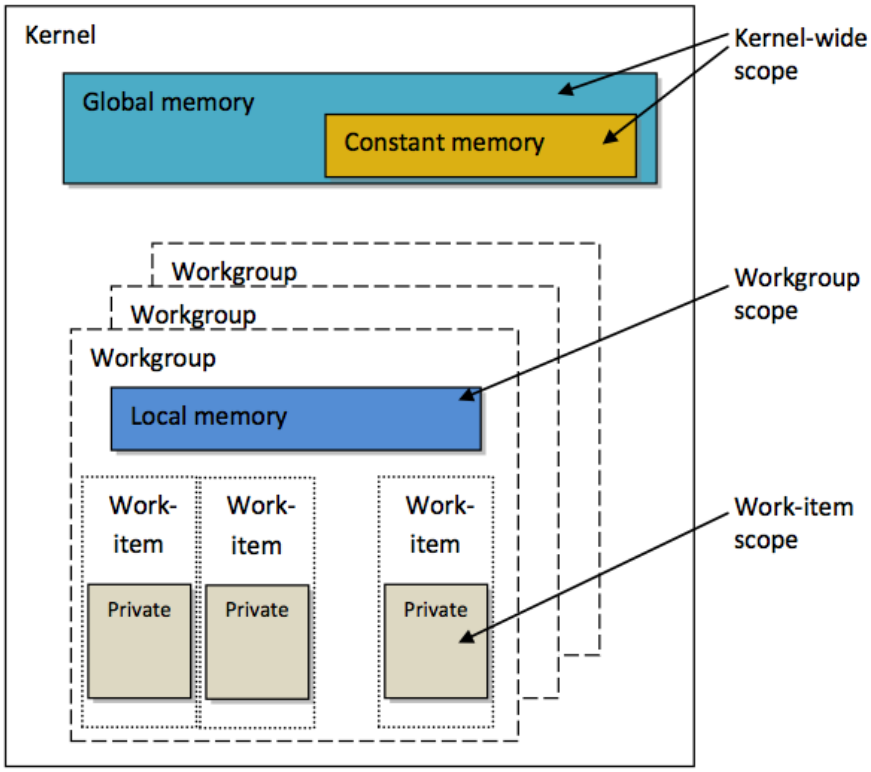
- Los work-items son equivalentes a los hilos en CUDA
- Los work-groups son equivalentes a los bloques en CUDA

Tipos de memoria:

- Memoria global, equivale a la memoria global de CUDA
- Memoria constante, sería como la memoria global de OpenCL pero de solo lectura
- Memoria local, equivale a la memoria compartida de CUDA
- Memoria privada, equivalente en CUDA a la memoria local por hilo

Modificadores: `__global`, `__constant`, `__local`, `__private`

Tipos de Memoria



Estructuras de Datos

Existen dos APIs (host), una para C y otra para C++

Platforma (`cl_platform_id`) - es una clase de dispositivos, como "intel cpu", "nvidia gpu" o "ati gpu"

Dispositivo (`cl_device_id`) - cada uno de los dispositivos que pueden ejecutar un *kernel*

Kernel (`cl_kernel`) - similar a los *kernels* de CUDA

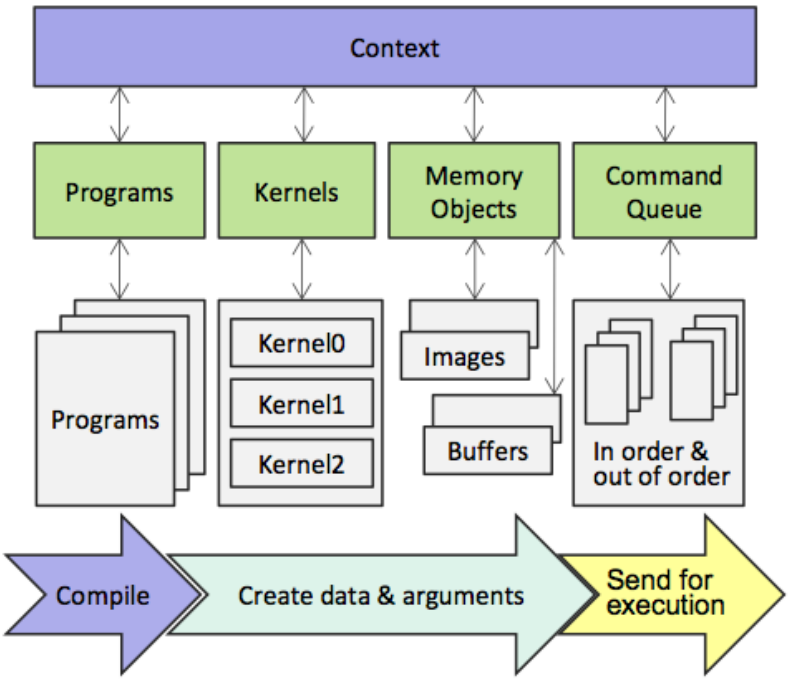
Programa (`cl_program`) - texto que contiene todos los *kernels*

Cola de órdenes (`cl_command_queue`) - es una cola de *kernels* que especifica el orden en que se han de ejecutar

Contexto (`cl_context`) - se asigna a uno o más dispositivos, permitiendo recibir *kernels* y transferir datos

Ejecución de Programas OpenCL

Es necesario añadir bastante código común en cualquier aplicación OpenCL



Principales Pasos

1. Obtener información acerca de dispositivos OpenCL disponibles
2. Crear un contexto para asociar dispositivos OpenCL
3. Crear programas para su ejecución en uno o más dispositivos asociados (compilación)
4. Seleccionar los *kernels* que se quieren ejecutar
5. Crear objetos de memoria accesibles en el *host*/*device*
6. Copiar datos al dispositivo
7. Lanzar los *kernels* en la cola de comandos para ejecutar
8. Copiar resultados desde el dispositivo

Ejemplo

Dado un vector, obtener otro vector con los elementos al cuadrado

Declaraciones

```
#include <CL/opencl.h>

...

  cl_int err;     // error code returned from api calls

  size_t global;  // global domain size for our calculation
  size_t local;   // local domain size for our calculation

  cl_platform_id platform;   // OpenCL platform
  cl_device_id device;       // compute device id
  cl_context context;        // compute context
  cl_command_queue commands; // compute command queue
  cl_program program;        // compute program
  cl_kernel kernel;          // compute kernel
```

Ejemplo: Paso 1

Obtener dispositivos

```
// Obtain the platform
err = clGetPlatformIDs(1, &platform, NULL);
if (err != CL_SUCCESS) {
  cerr << "Error: Failed to find a platform!" << endl;
  return EXIT_FAILURE;
}

// Get a device of the appropriate type
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) {
  cerr << "Error: Failed to create a device group!" << endl;
  return EXIT_FAILURE;
}
```

- En el caso general, tenemos un array de plataformas y varios dispositivos en cada una
- `CL_DEVICE_TYPE_ALL` para cualquier tipo

Ejemplo: Paso 2

Crear contexto

```
// Create a compute context
context = clCreateContext(0, 1, &device, NULL, NULL, &err);
if (!context) {
  cerr << "Error: Failed to create a compute context!" << endl;
  return EXIT_FAILURE;
}
```

El contexto podría englobar a más de un dispositivo

Crear cola de comandos

```
// Create a command commands
commands = clCreateCommandQueue(context, device, 0, &err);
if (!commands) {
  cerr << "Error: Failed to create a command commands!" << endl;
  return EXIT_FAILURE;
}
```

La cola se asigna a un dispositivo concreto del contexto

Ejemplo: Paso 3

Compilar programa

```
// Create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1,
                        (const char **) &KernelSource, NULL, &err);
if (!program) {
  cerr << "Error: Failed to create compute program!" << endl;
  return EXIT_FAILURE;
}

// Build the program executable
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
  size_t len;
  char buffer[2048];

  cerr << "Error: Failed to build program executable!" << endl;
  clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                        sizeof(buffer), buffer, &len);
  cerr << buffer << endl;
  exit(1);
}
```


Ejemplo: Paso 4

El programa está en una cadena

```
const char *KernelSource = \
  "__kernel void square(__global float* d_in,         \n" \
  "                     __global float* d_out, int N) \n" \
  "{                                                  \n" \
  "   int i = get_global_id(0);                       \n" \
  "   if (i<N) d_out[i] = d_in[i]*d_in[i];            \n" \
  "}                                                  \n";
```

Seleccionar el *kernel*

```
// Create the compute kernel in the program
kernel = clCreateKernel(program, "square", &err);
if (!kernel || err != CL_SUCCESS) {
  cerr << "Error: Failed to create compute kernel!" << endl;
  exit(1);
}
```

Ejemplo: Paso 5

Reservar memoria

```
// Allocate host data
float* h_data = new float[N];
float* h_res = new float[N];

// Fill the vector with random values
for (int i=0; i<N; i++) h_data[i] = rand()/(float)RAND_MAX;

// Create the device memory vectors
cl_mem d_in, d_out;
int nbyte = sizeof(float)*N;
d_in = clCreateBuffer(context, CL_MEM_READ_ONLY, nbyte, NULL, NULL);
d_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, nbyte, NULL, NULL);
if (!d_in || !d_out) {
  cerr << "Error: Failed to allocate device memory!" << endl;
  exit(1);
}
```


Ejemplo: Paso 6

Copiar datos

```
// Transfer the input vector into device memory
err = clEnqueueWriteBuffer(commands, d_in, CL_TRUE, 0,
                           nbyte, h_data, 0, NULL, NULL);
if (err != CL_SUCCESS) {
  cerr << "Error: Failed to write to source array!" << endl;
  exit(1);
}
```

También se podría haber copiado directamente al crear el *buffer*

```
flags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
d_in = clCreateBuffer(context, flags, nbyte, h_data, &err);
```

Ejemplo: Paso 7

Lanzar *kernel*

```
// Set the arguments to the compute kernel
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_in);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_out);
err |= clSetKernelArg(kernel, 2, sizeof(int), &N);
if (err != CL_SUCCESS) exit(1);

// Get maximum work group size for executing the kernel
err = clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_WORK_GROUP_SIZE,
                               sizeof(local), &local, NULL);
if (err != CL_SUCCESS) exit(1);

// Execute the kernel using the maximum number of work group items
global = N;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,
                             &local, 0, NULL, NULL);
if (err) return EXIT_FAILURE;

clFinish(commands);   // Wait for all commands to complete
```

`global` puede ser un array de 2 o 3 dimensiones
`local` puede omitirse (`NULL`), se eligen valores por defecto

Ejemplo: Paso 8

Copiar resultados

```
// Read back the results from the device to verify the output
err = clEnqueueReadBuffer( commands, d_out, CL_TRUE, 0, nbyte,
                           h_res, 0, NULL, NULL );
if (err != CL_SUCCESS) {
  cerr << "Error: Failed to read d_out array! " <<  err << endl;
  exit(1);
}
```

Finalmente se destruyen los objetos: `clReleaseKernel(kernel);` ...

Las llamadas de la cola de comandos permiten sincronizar utilizando el último argumento

```
cl_event event;
err = clEnqueueReadBuffer( commands, d_out, CL_FALSE, 0, nbyte,
                           h_res, 0, NULL, &event );
// Do useful work since read op is now non-blocking
err = clWaitForEvents(1, &event);
```

Compilación de *Kernels*

Compilación online

- El programa del *host* contiene el código fuente del programa del *device* (en una cadena o lo carga de disco)
- Se debe invocar al compilador de OpenCL durante la ejecución
- Esto garantiza la portabilidad

Compilación offline

- El *kernel* ya se encuentra compilado
- Útil en sistemas de tiempo real
- Portabilidad limitada, pero se puede proporcionar un binario por cada plataforma

En versiones recientes se puede usar SPIR: *Standard Portable Intermediate Representation*

Utilidades

En los *kernels* se dispone de funciones intrínsecas para cálculo de índices:

Función	Descripción
`get_work_dim`	Número de dimensiones en uso
`get_global_size`	Número de work items globales
`get_global_id`	Valor del ID global de work item
`get_local_size`	Número de work items locales
`get_local_id`	Valor del ID local de work item
`get_num_groups`	Número de work groups
`get_group_id`	ID de work group

Todas (excepto la primera) tienen un argumento `uint dimension`

```
int idx = get_global_id(0) + get_global_id(1) * get_global_size(0);
```

Sincronización

En los *kernels* lo más común es sincronizar *work-items* dentro del *work-group* en acceso coordinado a variables `__local`

```
barrier(CLK_LOCAL_MEM_FENCE);
```

También se puede usar `CLK_GLOBAL_MEM_FENCE` para sincronizar en el acceso a memoria global

Alternativas:

- Operaciones atómicas: `atomic_cmpxchg()`; también para reducciones: `atomic_add(), etc.`
- `mem_fence()`: garantiza la terminación de las operaciones de acceso a memoria anteriores al *fence* antes de las operaciones posteriores