# Arquitectura de Computadores

## Tema 5-5. Programación Paralela con CUDA C

# Sumario

- Introducción
- Vista del programador
- Código ejemplo
- Flujo de ejecución de un programa y asignación de recursos de GPU
- Jerarquía de hilos y la jerarquía de memoria asociada
- Evaluación de prestaciones

# Introducción

- CUDA es un lenguaje de programación de alto nivel parecido a C/C++ especializado en generar programas para GPU
  - OpenCL es otro lenguaje similar a CUDA pero independiente del fabricante del hardware
- Sistema computador: host + GPUs
- Código fuente `.cu`: declaraciones de datos + funciones GPU (*kernels*)
- Un programa CUDA tiene una parte para ser ejecutada en el procesador convencional (`__host__`) y otra parte en la GPU (`__device__`/`__global__`)
- Palabras clave CUDA
- Compilador de CUDA C (NVCC: NVIDIA C Compiler)

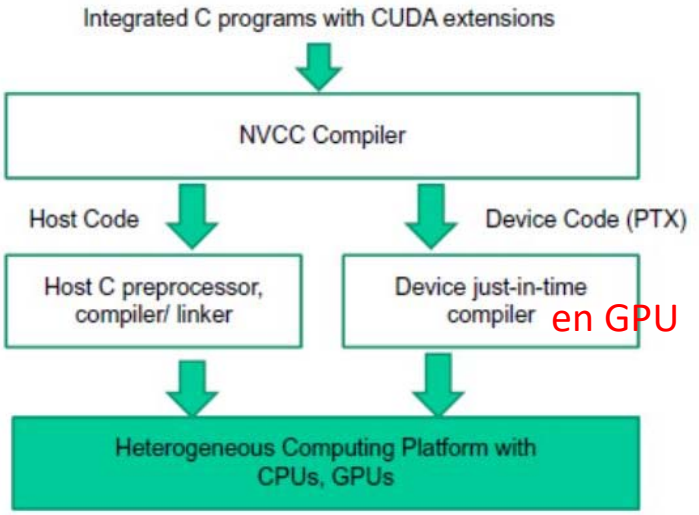
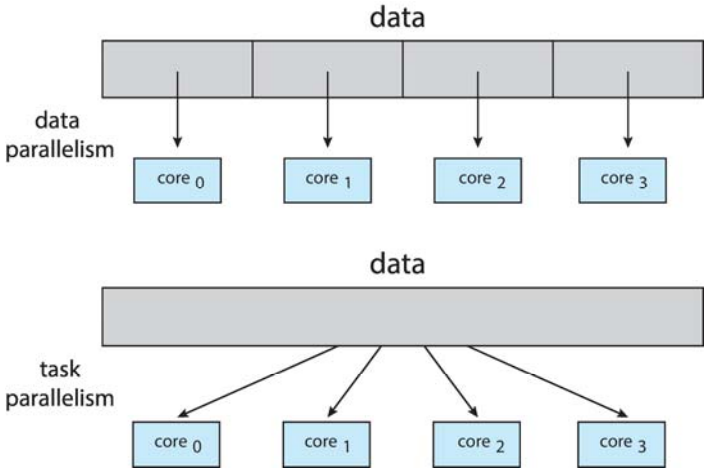
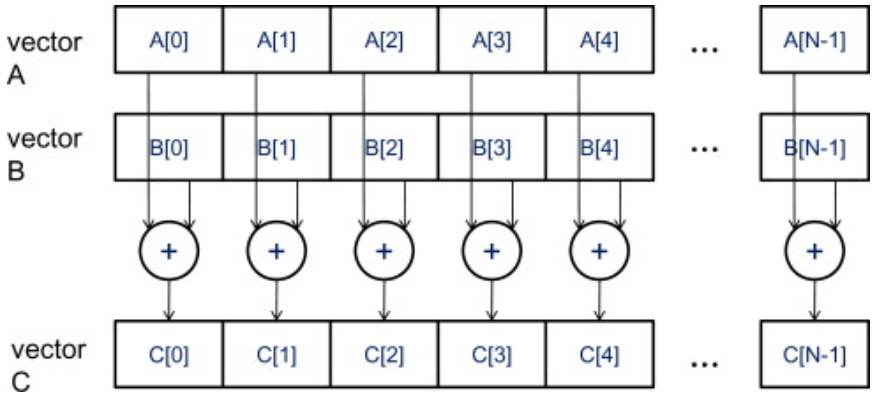



**FIGURE 3.2**
Overview of the compilation process of a CUDA program.

# Vista del programador de CUDA C

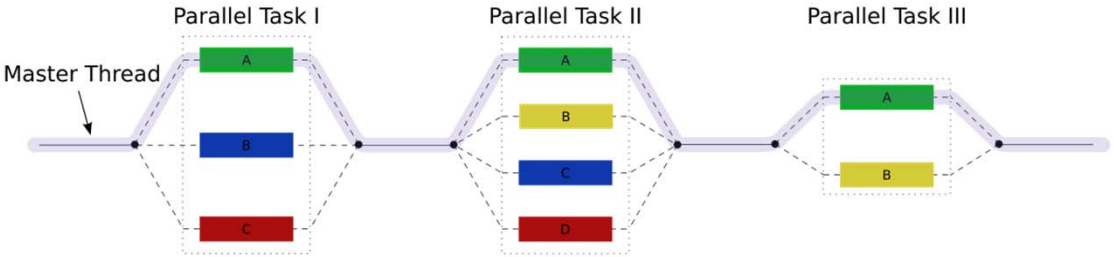
- CUDA C acelera las aplicaciones que muestran un alto paralelismo de datos (ejemplo: aplicaciones gráficas)
- CUDA C también puede acelerar aplicaciones con paralelismo MIMD de tareas
- Ejemplo: Paralelismo de datos en la suma de dos vectores

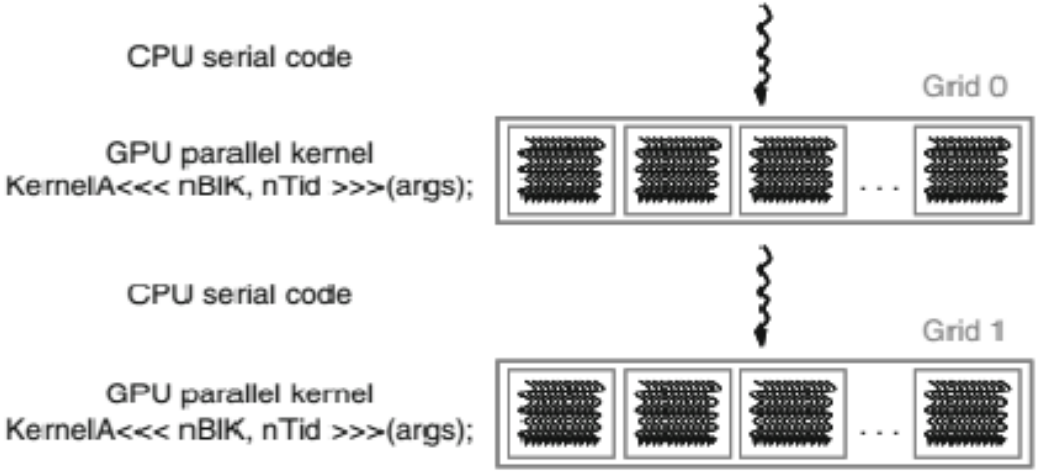



# Ejecución de un programa CUDA C

- *Hilo* CUDA: primitiva de paralelismo en CUDA
- *Bloque* de hilos CUDA: conjunto de 32 hilos que se ejecutan en paralelo en un multiprocesador de la GPU
- Cuando se llama a un *kernel*, se ejecuta un *grid* que consiste en varios *bloques* de *hilos* en los multiprocesadores
- Los hilos se generan y lanzan a ejecutar por dispositivos GPU que se denominan *planificadores*. *Tarda poco tiempo.*
- Los bloques de hilos pretenden explotar el paralelismo de datos SIMD
- La ejecución en host y GPU podría estar solapada aunque en el gráfico no se muestre así



# Código ejemplo en C

3 variables pasadas por referencia de la dirección de memoria (punteros) donde comienzan los vectores en memoria, 2 de entrada, 1 de salida

h_* : variable en el host

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

# Código ejemplo en CUDA C
# (escalable: para distintos tipos de GPUs)

```
#include <cuda.h>
```

declaración de funciones y variables propias de CUDA C

```
void vecAdd(float* h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float); // Size in bytes
  float *d_A, *d_B, *d_C; // Device pointers

  // Device memory allocation
  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);

  // Host->device memory transfer
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Kernel launch
  vecAdd_kernel<<<(n + 256 - 1) / 256, 256>>>(d_A, d_B, d_C, n);

  // Device->Host memory transfer
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Device memory deallocation
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

tamaño de los vectores

d_* : variable en el GPU

reserva espacio de las variables en GPU

copia las variables desde el host a la GPU

ejecuta el kernel en la GPU

copia el vector resultado desde GPU al host

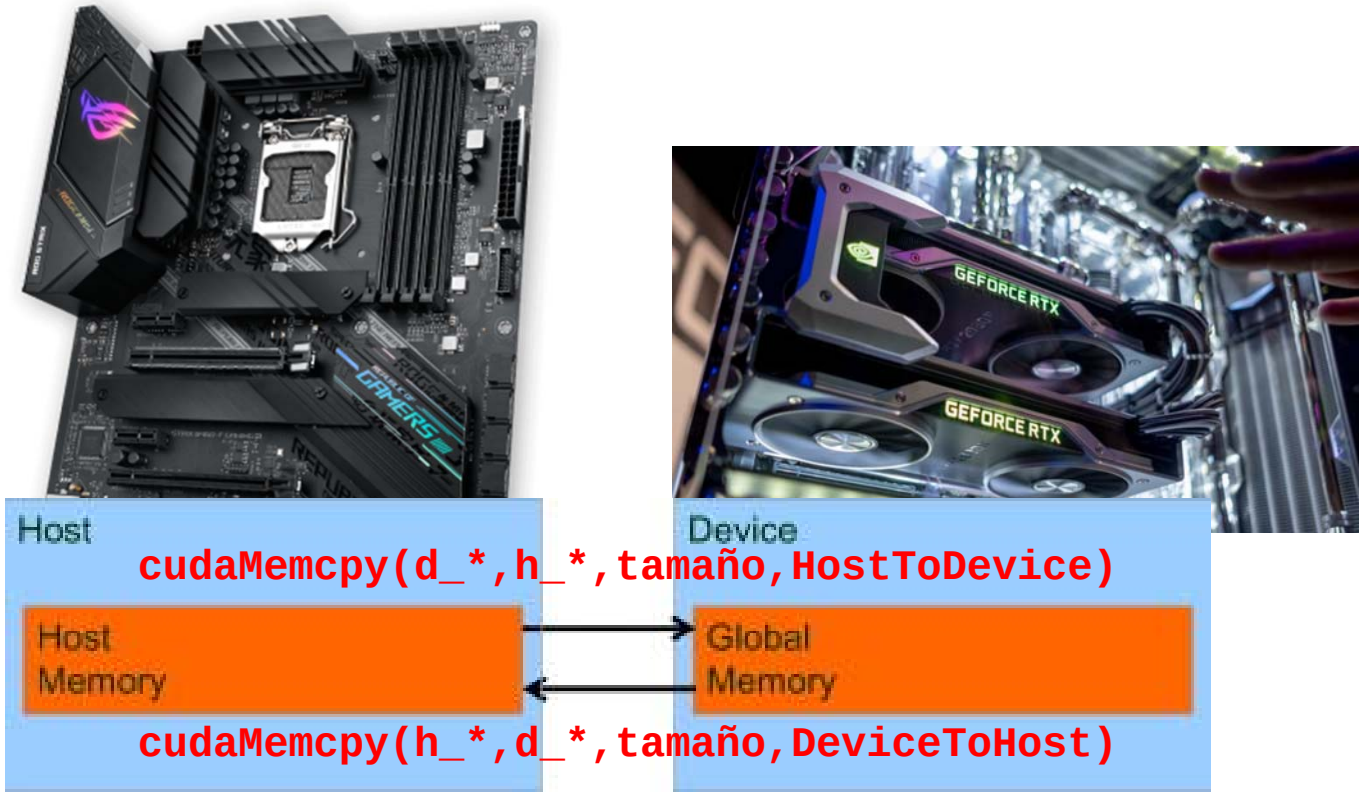
libera el espacio de memoria reservado previamente

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

**FIGURE 3.11**

A vector addition kernel function and its launch statement.

# Envío y recepción de variables del programa



# Código ejemplo en CUDA C

```
void vecAdd(float* h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float); // Size in bytes
  float *d_A, *d_B, *d_C; // Device pointers

  // Device memory allocation
  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);

  // Host->device memory transfer
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Kernel launch
  vecAdd_kernel<<<(n + 256 - 1) / 256, 256>>>(d_A, d_B, d_C, n);

  // Device->Host memory transfer
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Device memory deallocation
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

kernel que genera el cómputo de cada hilo
→ **no hay definición de bucles**
→ **bucle se remplaza por un grid de hilos**

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

ejecutable en la GPU

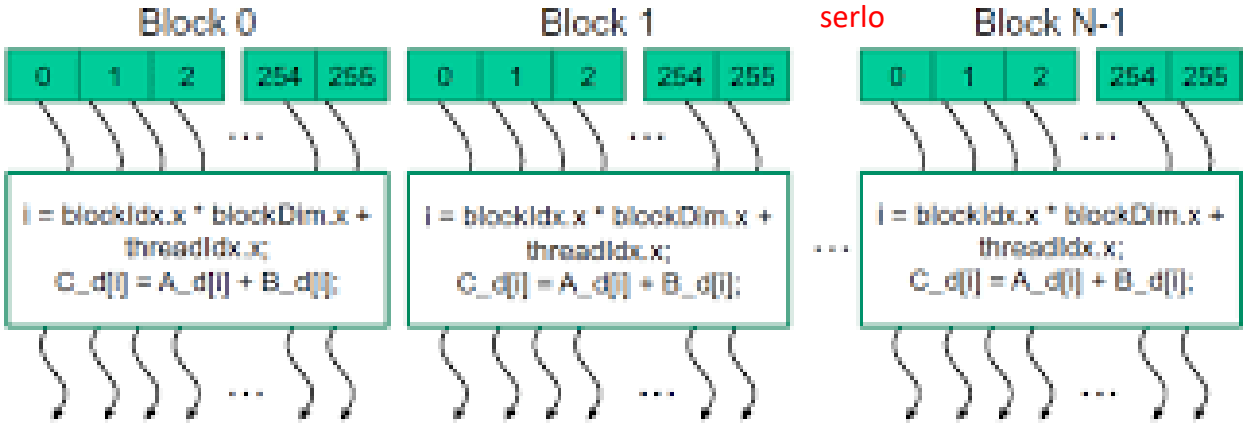
Variables CUDA

variable local del hilo, índice global de cada hilo:
**`hilo[blockIdx.x, threadIdx.x]`**

Número de bloques de hilos en el grid

256 Hilos/bloque

Porque el número de hilos es proporcional a 32 y n no tiene que serlo




Genera un grid de hilos organizados en 2 niveles.
**Nivel 1:** el grid es organizado en un array de bloques de hilos; cada bloque tiene el mismo número de hilos (número de hilos en un bloque: `blockDim.x`) y tiene un identificador (`blockIdx.x`); kernel es lanzado a ejecutar con un número determinado de hilos.
**Nivel 2:** cada hilo tiene un identificador dentro del bloque (`threadIdx.x`)

# Código fuente

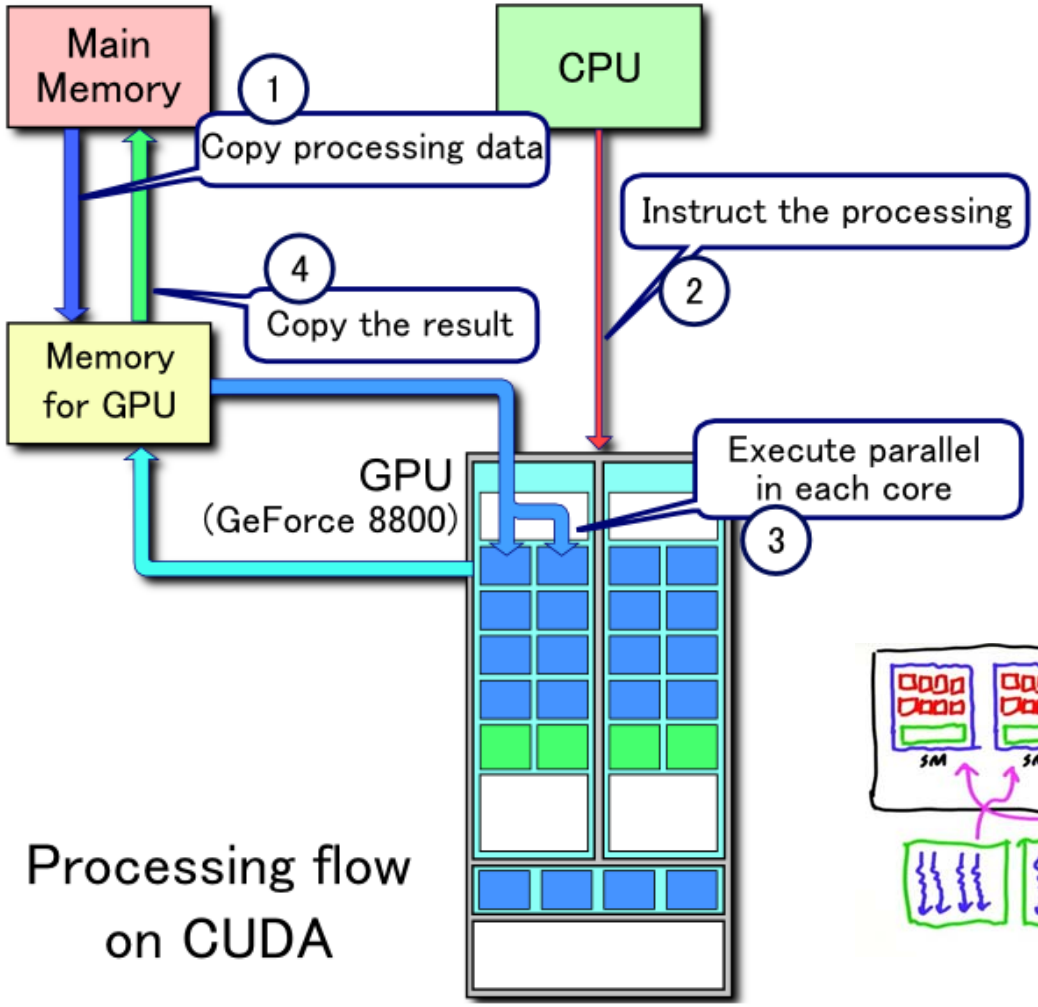
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd
(double *a, double *b, double *c, int n)
{
   // Get our global thread ID
   int id = blockIdx.x*blockDim.x+threadIdx.x;

   // Make sure we do not go out of bounds
   if (id < n)
      c[id] = a[id] + b[id];
}
```

```
int main( int argc, char* argv[] )
{
   // Size of vectors
   int n = 100000;

   // Host input vectors
   double *h_a;
   double *h_b;
   //Host output vector
   double *h_c;

   // Device input vectors
   double *d_a;
   double *d_b;
   //Device output vector
   double *d_c;

   // Size, in bytes, of each vector
   size_t bytes = n*sizeof(double);
```

```
// Allocate memory for each vector on host
   h_a = (double*)malloc(bytes);
   h_b = (double*)malloc(bytes);
   h_c = (double*)malloc(bytes);

   // Allocate memory for each vector on GPU
   cudaMalloc(&d_a, bytes);
   cudaMalloc(&d_b, bytes);
   cudaMalloc(&d_c, bytes);

// Initialize vectors on host
   for( int i = 0; i < n; i++ ) {
      h_a[i] = sin(i)*sin(i);
      h_b[i] = cos(i)*cos(i);
   }

   // Copy host vectors to device
   cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
   cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

// Number of threads in each thread block
   int blockSize = 1024;

   // Number of thread blocks in grid
   int gridSize = (int)ceil((float)n/blockSize);
```

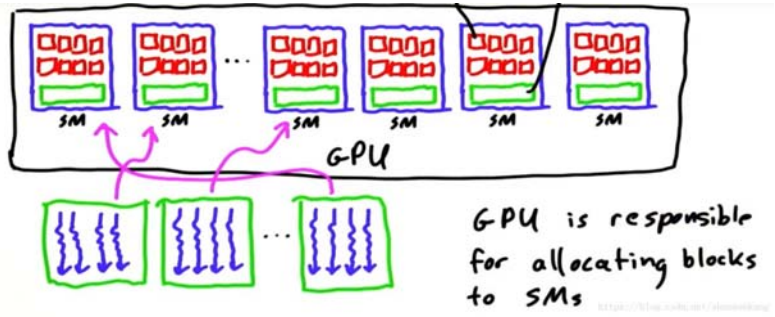
```
// Execute the kernel
   vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

   // Copy array back to host
   cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

// Release device memory
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);

   // Release host memory
   free(h_a);
   free(h_b);
   free(h_c);

   return 0;
}
```

# Flujo de ejecución de un programa y asignación de recursos en GPU (el asociado a CUDA C)




Cuando se envía a ejecutar el kernel, el sistema **"CUDA runtime"** genera el grid de hilos

La asignación de recursos hw se realiza bloque a bloque. Cada bloque de hilos se ejecuta en un multiprocesador



# Compilación con **nvcc** y ejecución

```
$ nvcc -c vecAdd.cu
$ nvcc -o vecAdd vecAdd.o -lcuda

$ ./vecAdd
```

# Jerarquía de hilos en CUDA C

- Llamada estándar a kernel:

```
dim3 dimBlock(128,1,1);
dim3 dimGrid(32,1,1);
nombreKernel<<<dimGrid, dimBlock>>>(parametros);
```

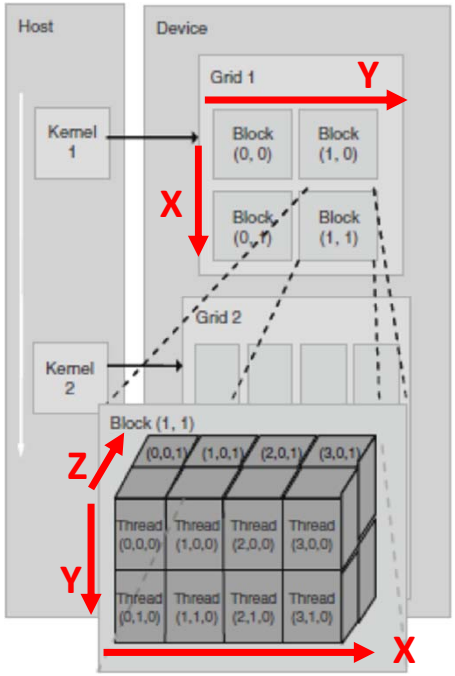
Estructuras de datos 3D que el programador define para establecer la jerarquía de niveles de hilos:
- organización de bloques del grid
- organización de hilos en cada bloque
- Número total de hilos= 32 x 128 = 4096 hilos

  - `dimGrid`: dimensiones 3D del código en bloques de hilos
    - Posibles valores: {`dimGrid.x`, `dimGrid.y`, `dimGrid.z`}= 1 … 65536
  - `dimBlock`: dimensiones 3D del bloque en hilos
    - Posibles valores: `dimBlock.x+dimBlock.y+dimBlock.z` <= 1024

```
nombreKernel<<<32, 128>>>(parametros);
```

    - Llamada equivalente para estructuras 1D
  - Otro ejemplo:
    - dim3 dimBlock(2,2,1);
    - dim3 dimGrid(4,2,2);
    - Kernel1<< <dimGrid, dimBlock> >>(. . .);

2 x 2 hilos en cada bloque
4 x 2 x 2 bloques de hilos en cada grid
Número total de hilos= 4 x 16= 64 hilos



# CUDA
# (*Compute Unified Device Architecture*)

- Jerarquía de memoria ~ Jerarquía de hilos

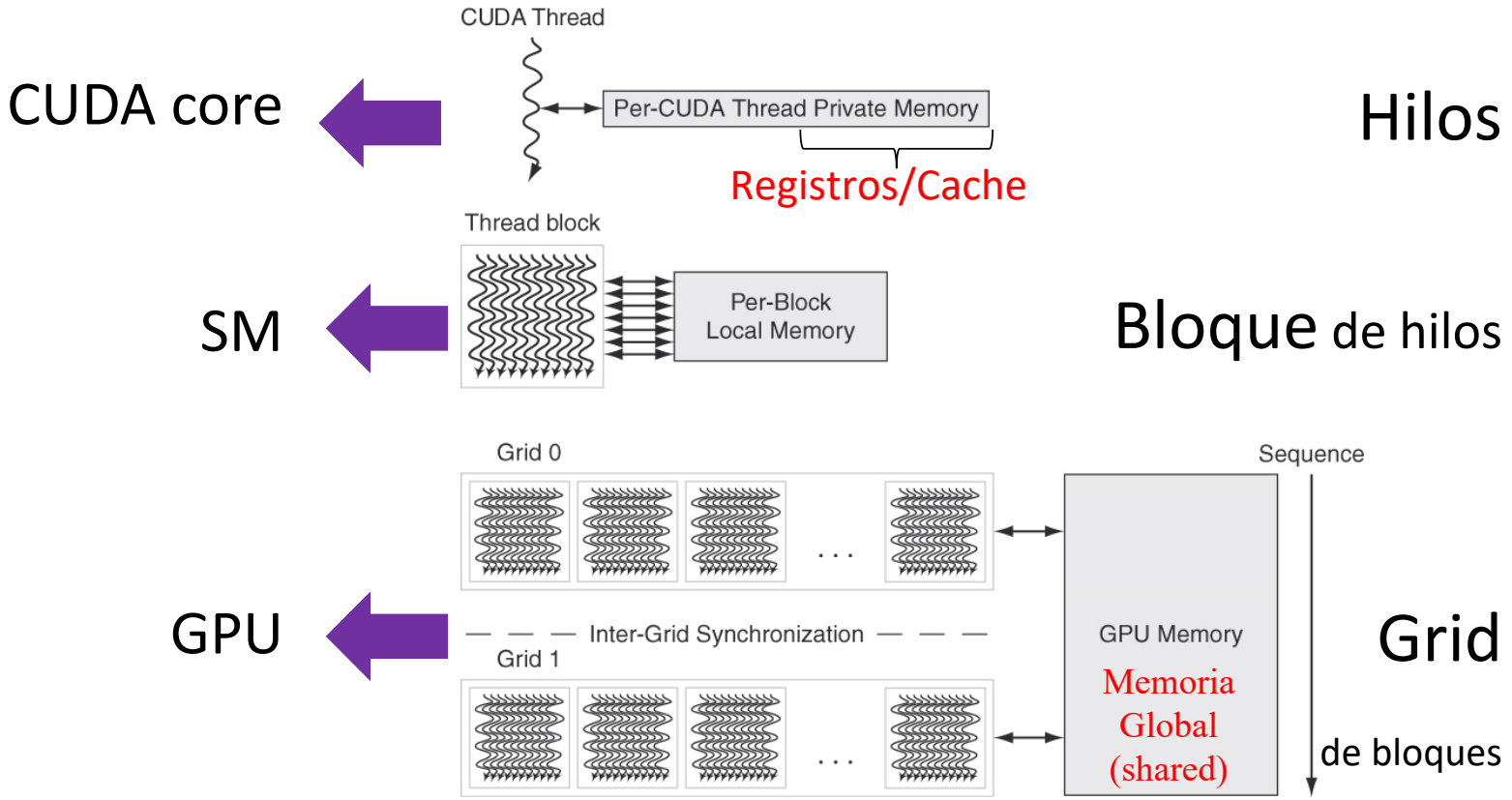



Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

# Ejemplo de ejecución de programa CUDA: `Multiplicación de 2 matrices n x n`

Versión secuencial C

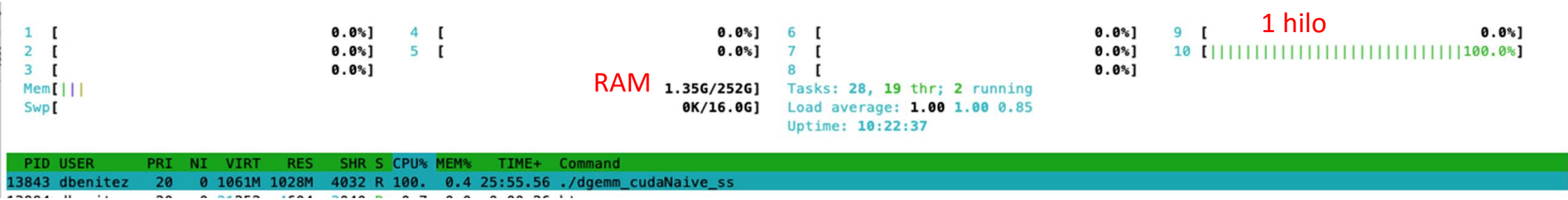
```
void dgemm_secuencial (int n, float* A, float* B, float* C)
{
 int i,j,k;
 for (i = 0; i < n; i++)          /* i: columna de C */
   for (j = 0; j < n; j++)        /* j: fila de C */
   {
    float cji = 0;                /* cji = C[j][i] */
    for(k = 0; k < n; k++ )
      cji += B[k+j*n] * A[i+k*n]; /* cji += B[j][k]*A[k][j] */
    C[i+j*n] = cji;               /* C[j][i] = cji */
   }
}
```

Versión CUDA C

```
__global__ void dgemHilos (int n, float* A, float* B, float* C){
int j = blockIdx.y * blockDim.y + threadIdx.y;    // Fila j de C
int i = blockIdx.x * blockDim.x + threadIdx.x;    // Columna i de C
int k;
float cji;
cji = 0;                                  /* cji = C[j][i] */
for (k = 0 ; k < n ; k++){
        cji += B[k+j*n] * A[i+k*n]; /* cji += B[j][k]*A[k][i] */
 }
C[i+j*n] = cji;
}
```

# Evaluación de prestaciones

**[htop]** Versión secuencial: Intel Xeon E5-2630 v4 @ 2.20GHz




Salida del programa

Multiplicacion de 3 matrices de 8192 x 8192 floats

Inicializacion de las matrices: 0.62 seg

**dgemm Secuencial: 6457.37 seg 170.3 MFlops**

**1,8 horas**

----------Informacion GPU-CUDA --------
CUDA Device Name: Tesla K40m
CUDA Capability: 3.5
CUDA maxThreadsPerBlock: 1024
CUDA maxThreadsDim, X: 1024, Y: 1024, Z: 64
CUDA maxGridSize, X: 2147483647, Y: 65535, Z: 65535
CUDA sharedMemPerBlock: 49152 Bytes
CUDAtotalConstantMemory: 65536 Bytes
CUDA SIMDWidth: 32
CUDA regsPerBlock: 65536
CUDA frecuencia reloj: 745000 KHz

# Evaluación de prestaciones

Salida del programa

```
----------Informacion Multiplicacion de Matrices --------
Numero total de elementos de cada matriz: 8192 x 8192 =
67108864
Tamanyo block-GPU: 32 x 32 = 1024 hilos/block
Tamanyo grid-GPU: 65536 blocks/grid
Dimensiones grid-GPU, X: 256 blocks, Y: 256 block

---------- TIEMPOS en GPU --------
dgemm_GPU-initGPU:              0.61 seg
dgemm_GPU-informacionGPU:  0.00 seg
dgemm_GPU-reservaMemGPU: 0.10 seg
dgemm_GPU-copiaMemGPU:     0.06 seg
dgemm_GPU-procesaGPU:         8.15514 seg
dgemm_GPU-copiaCPU:            0.03 seg
dgemm_GPU-freeGPU:             0.05 seg
```

**dgemm_GPU-TOTAL:  9.00 seg, 122175.8 MFlops**

**Speed-Up= 6457 / 9 = 717,5**