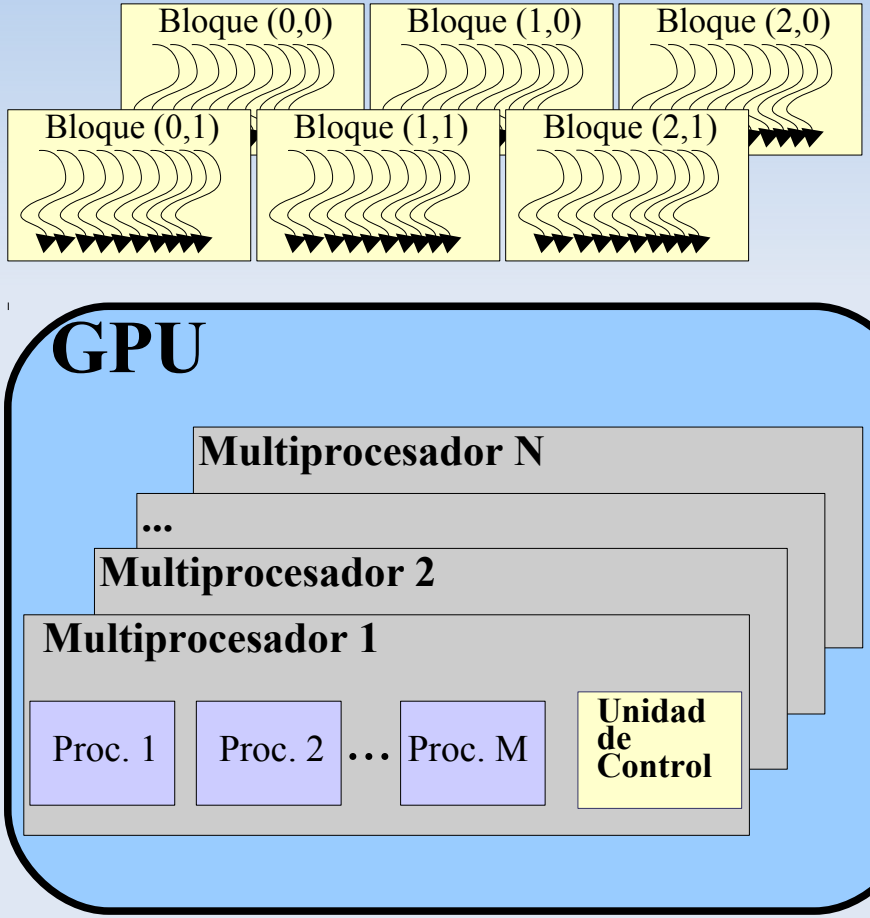
Introducción a la programación de hardware gráfico paralelo:CUDA



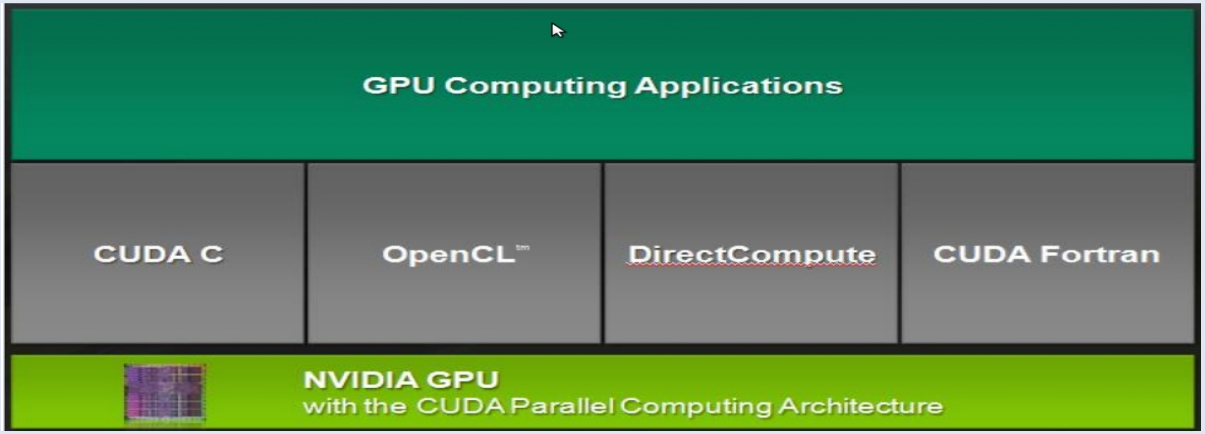
kubuntu

Introducción a CUDA 3.0

- Compute Unified Device Architecture (CUDA)

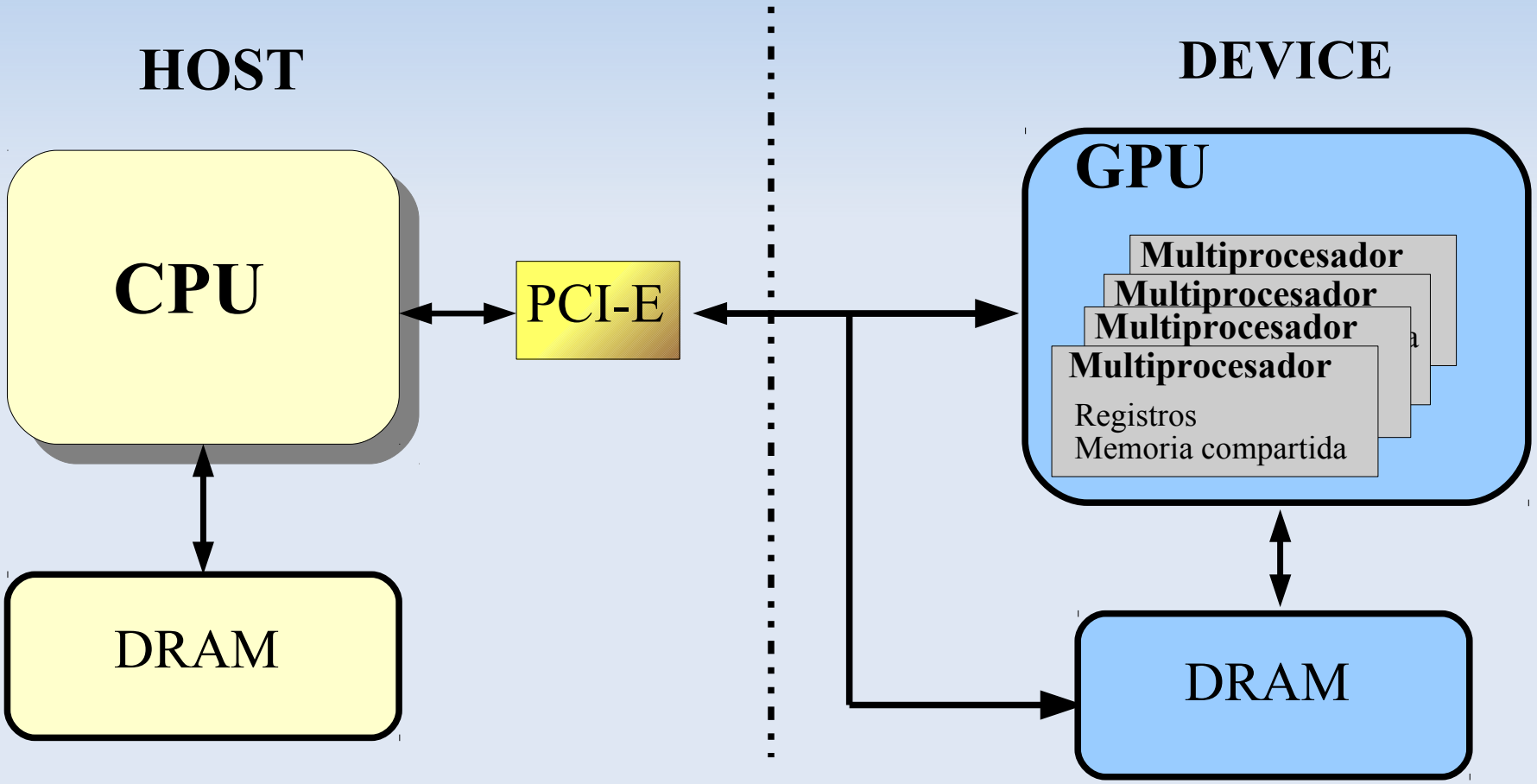
Plataforma hardware-software que permite aprovechar de forma eficiente el potencial de las GPUs de NVIDIA para resolver problemas muy costosos

- Entorno software asequible: Incluye extensión de C para programación de GPUs (CUDA C). También soporta otros lenguajes y APIs



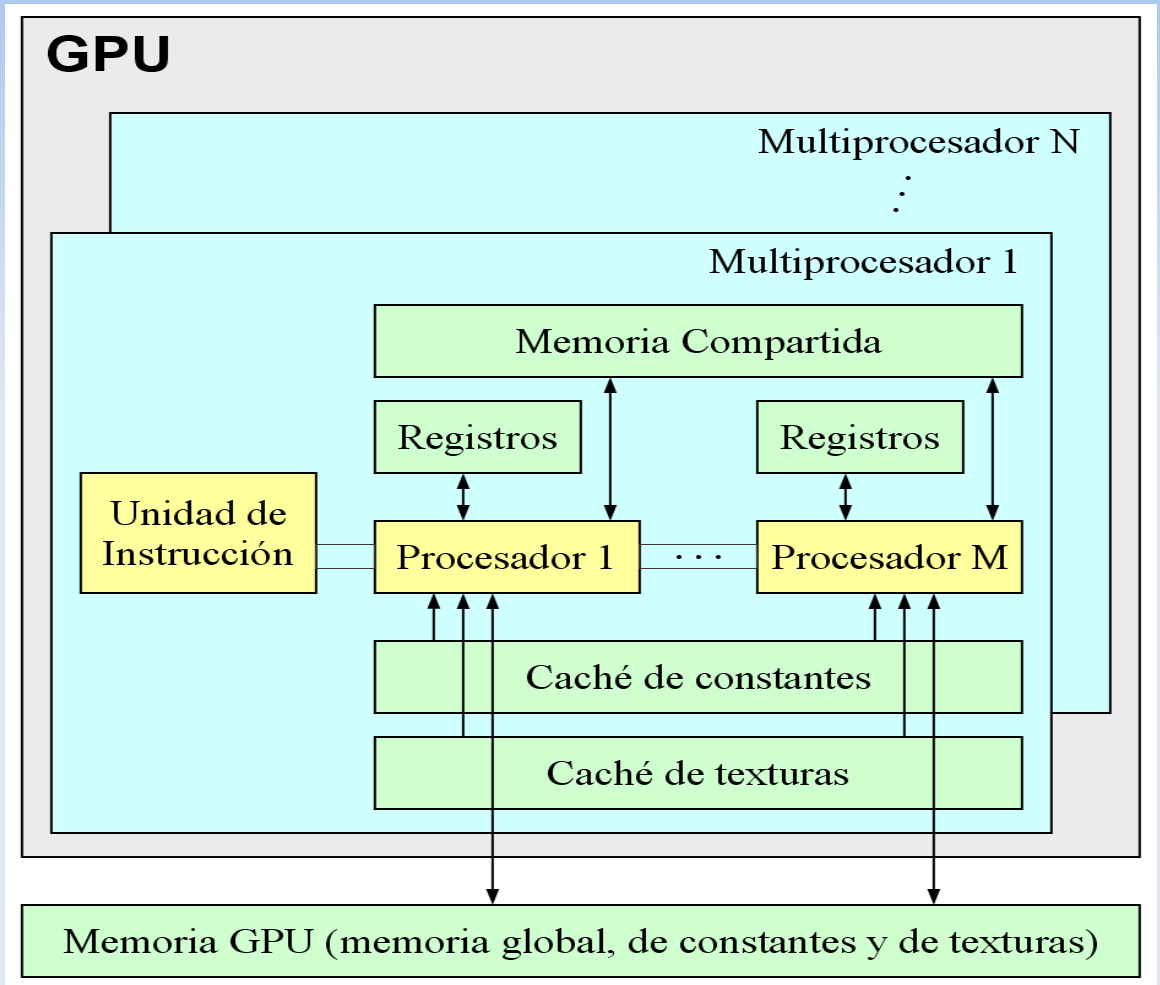
GPU como coprocesador

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



Modelo hardware (GPU)

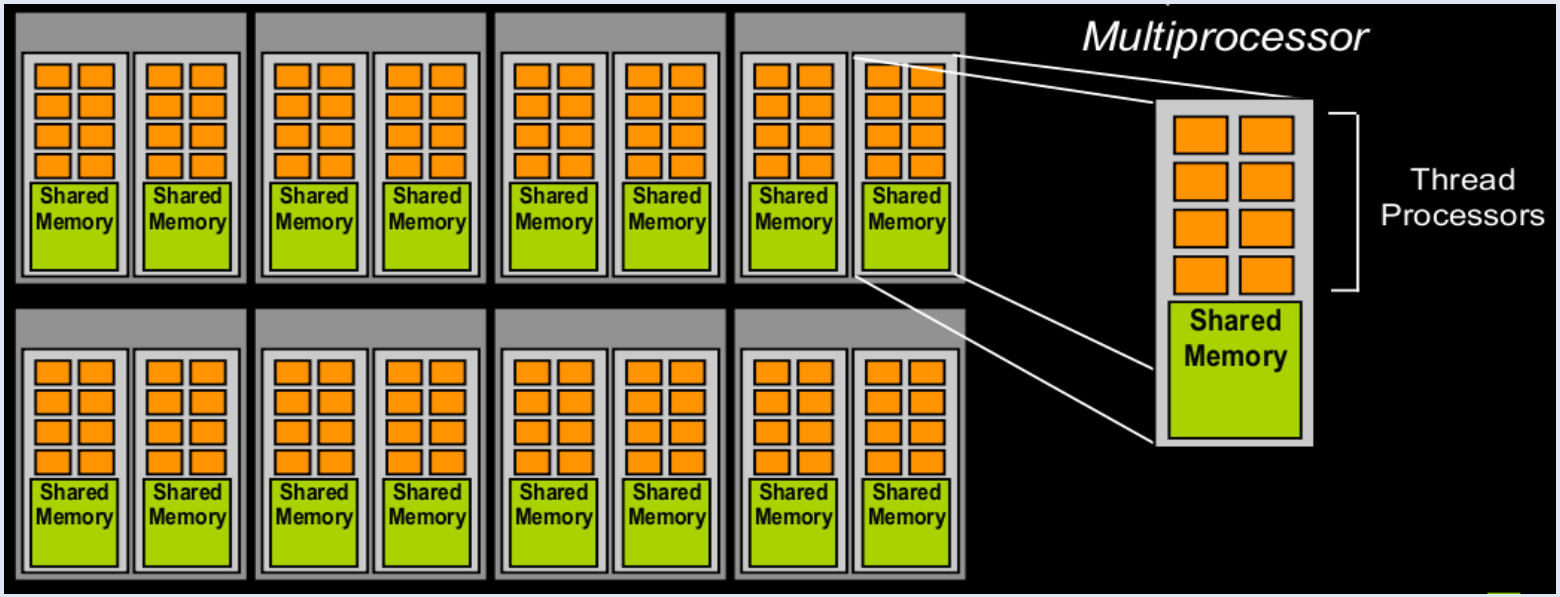
- LA GPU contiene **N multiprocesadores**.
- Cada multiprocesador incluye:
  - **M procesadores**
  - Banco de **registros**
  - **Memoria compartida**: muy rápida, pequeña.
  - **Cachés** de ctes y de texturas (sólo lectura)
- La **memoria global** es 500 veces más lenta que la memoria compartida



Arquitectura hardware

Serie 8 (G80)

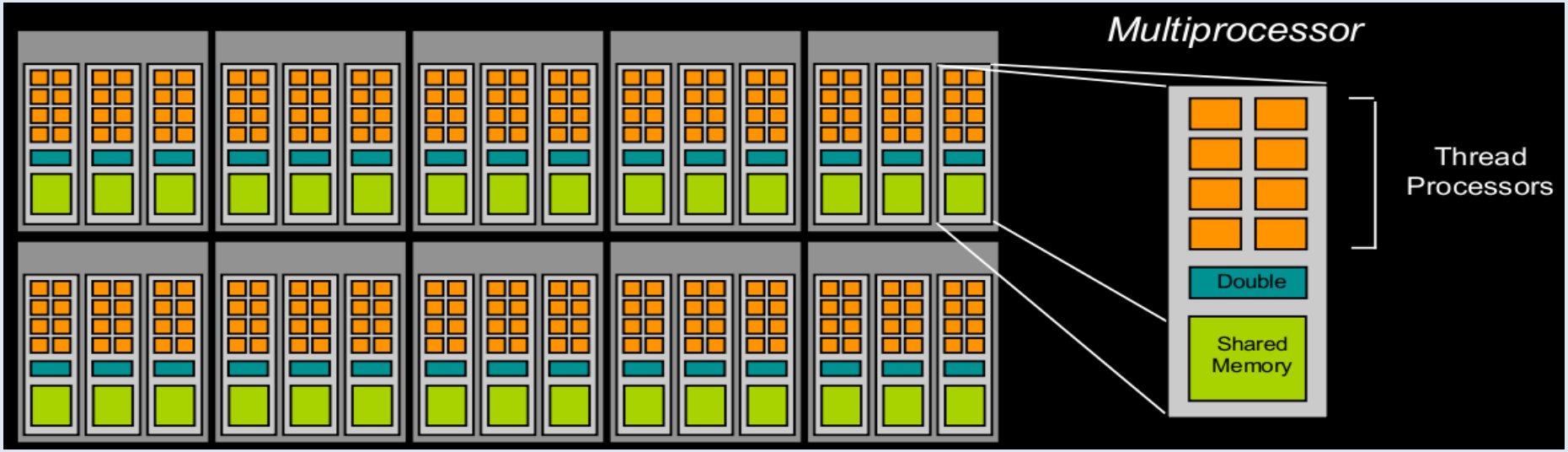
- 128 procesadores de hebras
- 16 multiprocesadores cada uno con 8 procesadores de hebras y memoria compartida de 16KB.



Arquitectura hardware

Serie 200 (GT200)

- 240 procesadores de hebras
- 30 multiprocesadores cada uno con 8 procesadores de hebras, una unidad de doble precisión y memoria compartida de 16KB.



Arquitectura hardware

Fermi (GT400)

- 512 cores
- 16 multiprocs.
- Cada multiprocs. con 32 cores y 16 Unidades de doble precisión.
- 64 KB. de SRAM a repartir entre memoria compartida y Cache L1.

kubuntu

Kernels CUDA

- Kernel en CUDA C: Función C que se ejecutará N veces en paralelo por N hebras CUDA diferentes.

- No se pueden ejecutar varios kernels en paralelo en el mismo device.

- **Restricciones**:
  - No puede acceder a memoria de host
  - Devuelve tipo void
  - Nº argumentos no variable
  - No recursivo
  - Sin variables static

- Se declaran usando modificador

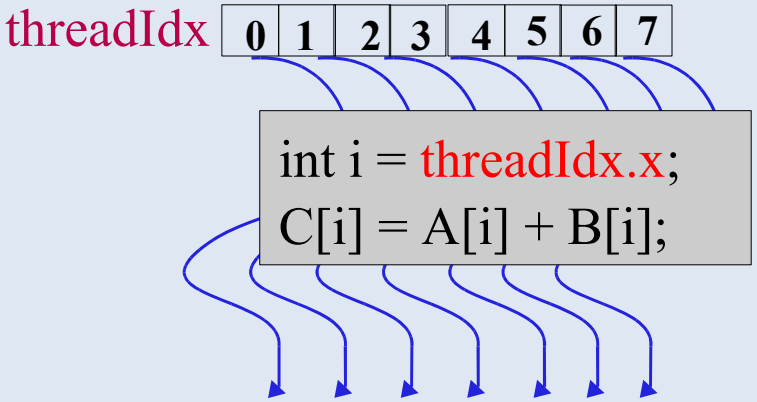
__global__

Suma de dos vectores Ay B de N floats

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

Hebras CUDA

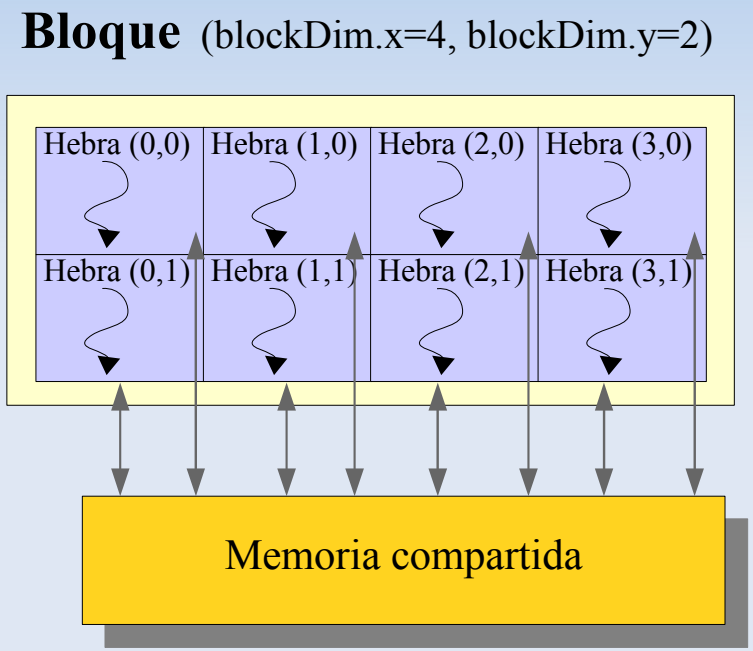
- **Extremadamente ligeras:**
  - Creación y cambios de contexto rapidísimos
  - Para lograr eficiencia: Descomposiciones de grano fino que permitan lanzar miles de hebras.
- Un Kernel se ejecuta por un **array de hebras**
  - Todas ejecutan el mismo código pero la acción depende del Identificador de hebra



•El identificador de hebra **threadIdx** (de tipo **uint3**) se usa para calcular direcciones de memoria y tomar decisiones de control

Bloques de hebras

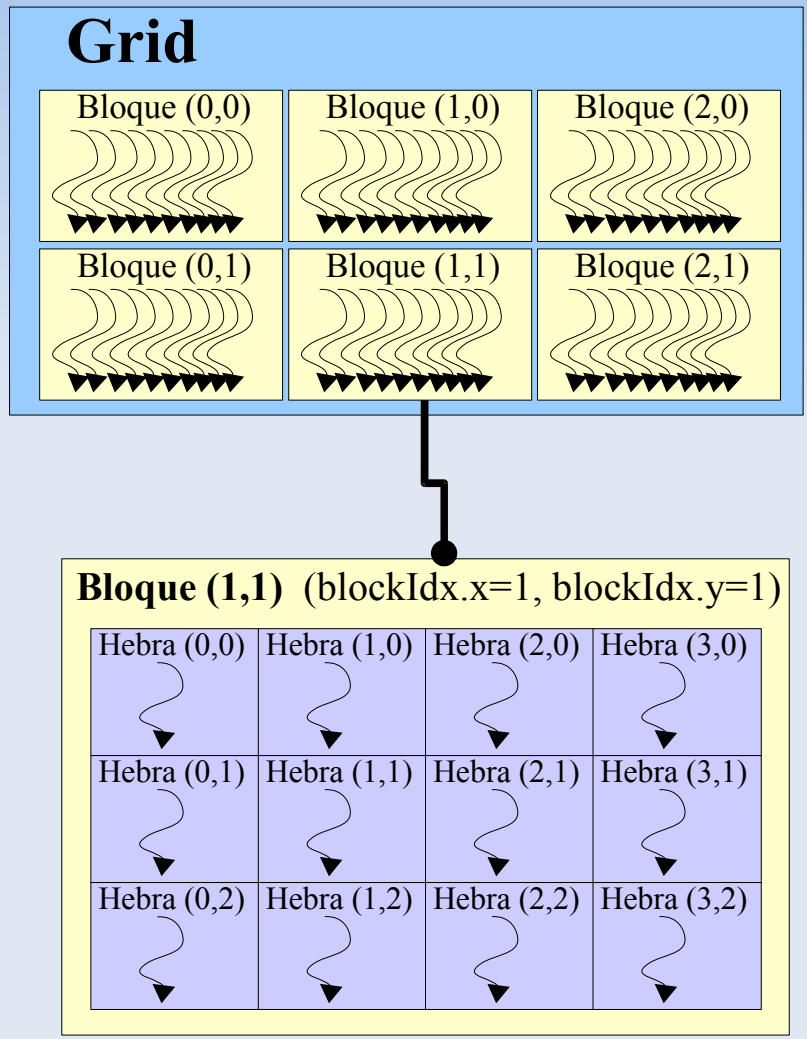
- Cada hebra se identifica con un índice de hebra (threadIdx) unidimensional, bidimensional o tridimensional dentro de un **bloque de hebras**.
  - Unidad de asignación de hebras a multiprocesadores.
  - Cada multiprocesador puede tener asignados 8 bloques y cada bloque puede incluir hasta 512 hilos.
  - Las hebras de un bloque pueden comunicarse a través de memoria compartida.
  - La variable predefinida **blockDim** (de tipo **dim3**) permite acceder a las dimensiones del bloque dentro de un kernel



kubuntu

Grid de hebras

- **Grid de bloques**: Los bloques se organizan formando una matriz 1D o 2D denominada grid.
  - El número de hebras por bloque y bloques por grid que se usarán para lanzar un kernel se especifica con la sintaxis <<<..., ...>>>.
- **Variables predefinidas**
  - **uint3 blockIdx:** permite identificar el bloque dentro del grid.
  - **dim3 gridDim:** guarda las dimensiones del grid.
- Las dimensiones del grid y el bloque para un kernel deben ser escogidas por el programador para maximizar eficiencia.

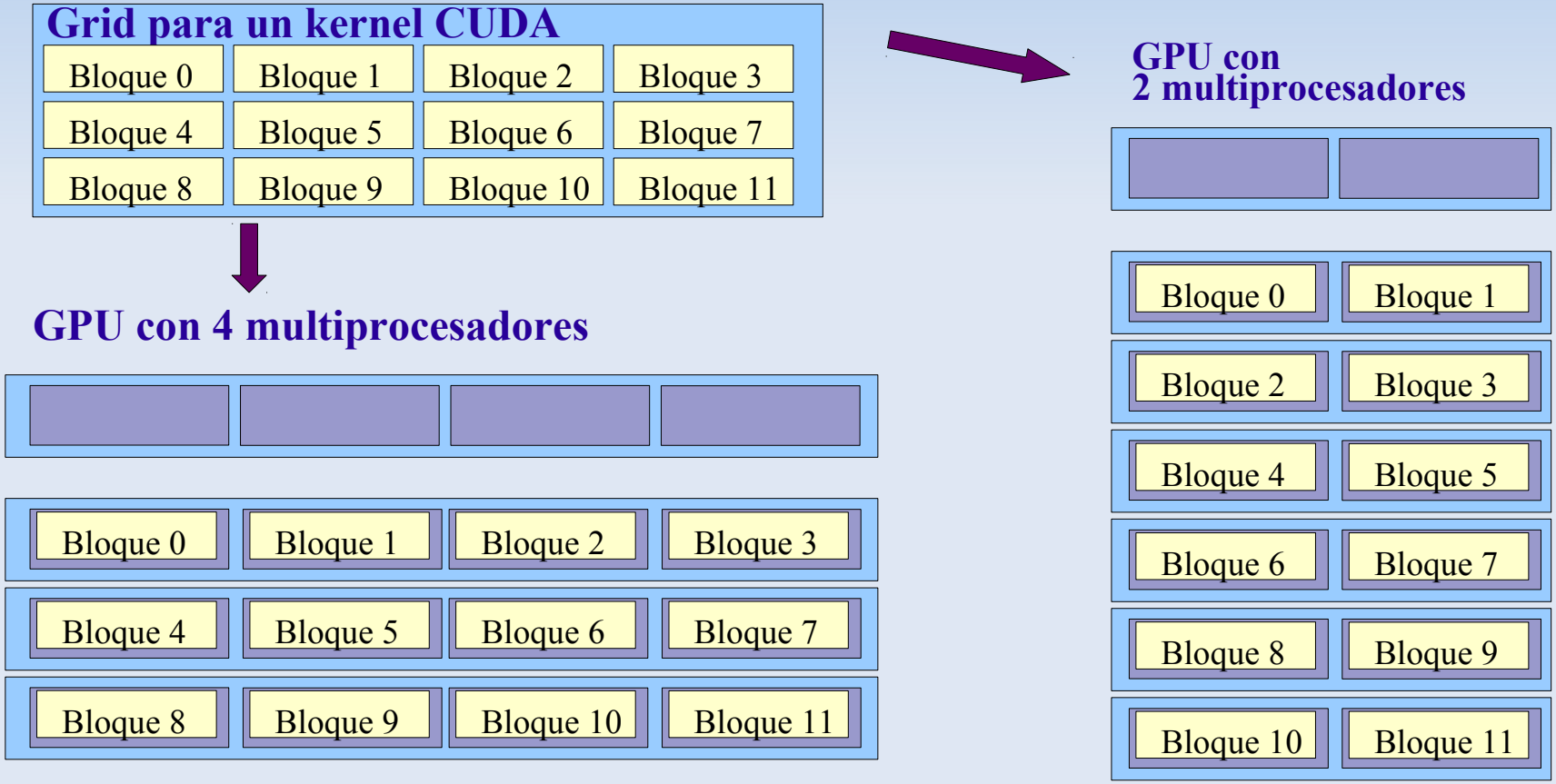


Suma de matrices N×N en GPU

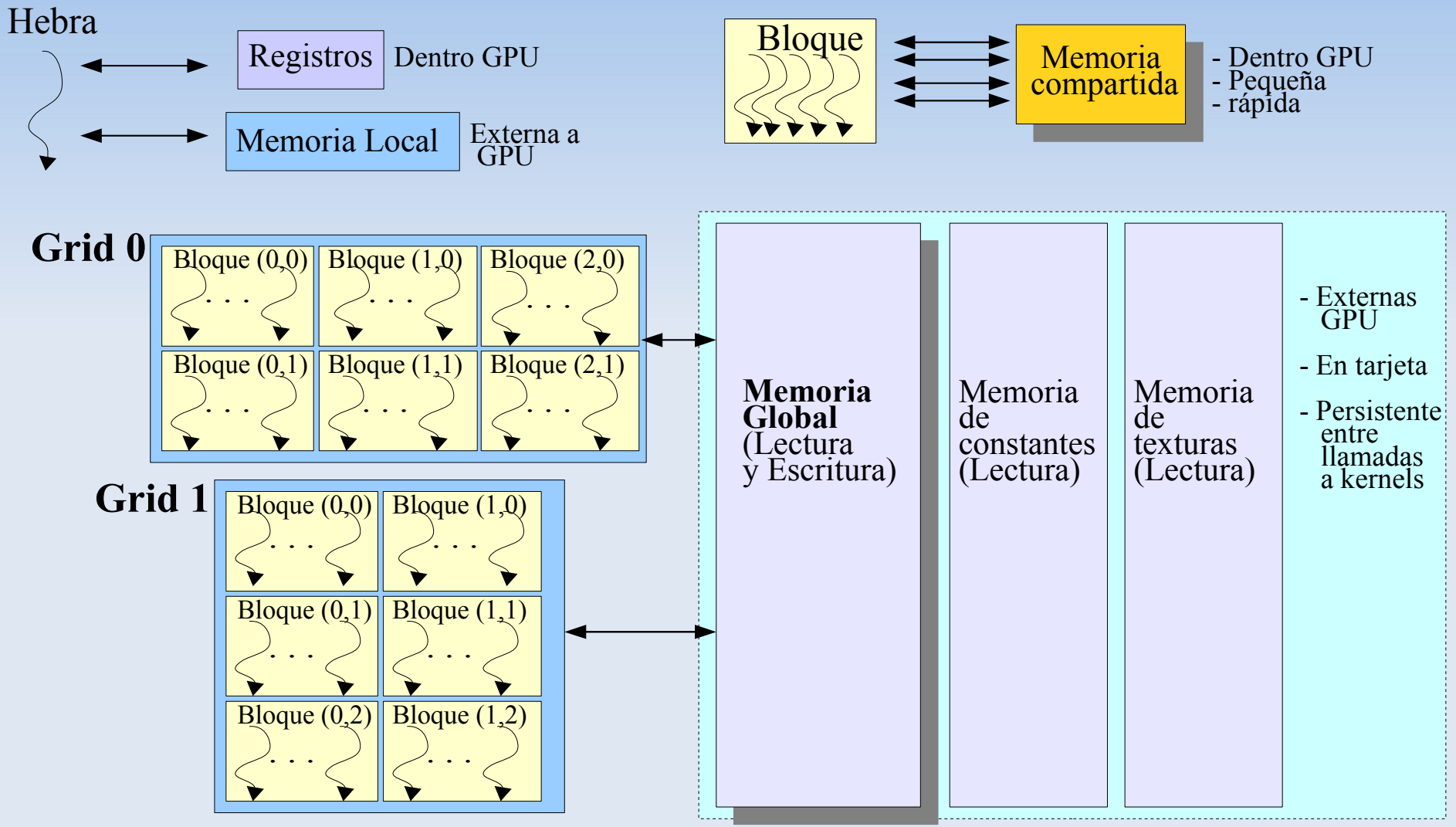
```
__global__ void MatAdd (float A[N][N], float B[N][N],float C[N][N])
        {
          int i = blockIdx.x * blockDim.x + threadIdx.x;
          int j = blockIdx.y * blockDim.y + threadIdx.y;
          if (i < N && j < N)   C[i][j] = A[i][j] + B[i][j];
        }

int main()
        {
          ...
          // Invocación de un 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 de hebras a multiprocesadores: Los kernels escalan a distinto número de MPs.



Jerarquía de Memoria



kubuntu

Programación heterogénea

Programa C

__global__ Kernel0 (...)
{ … }

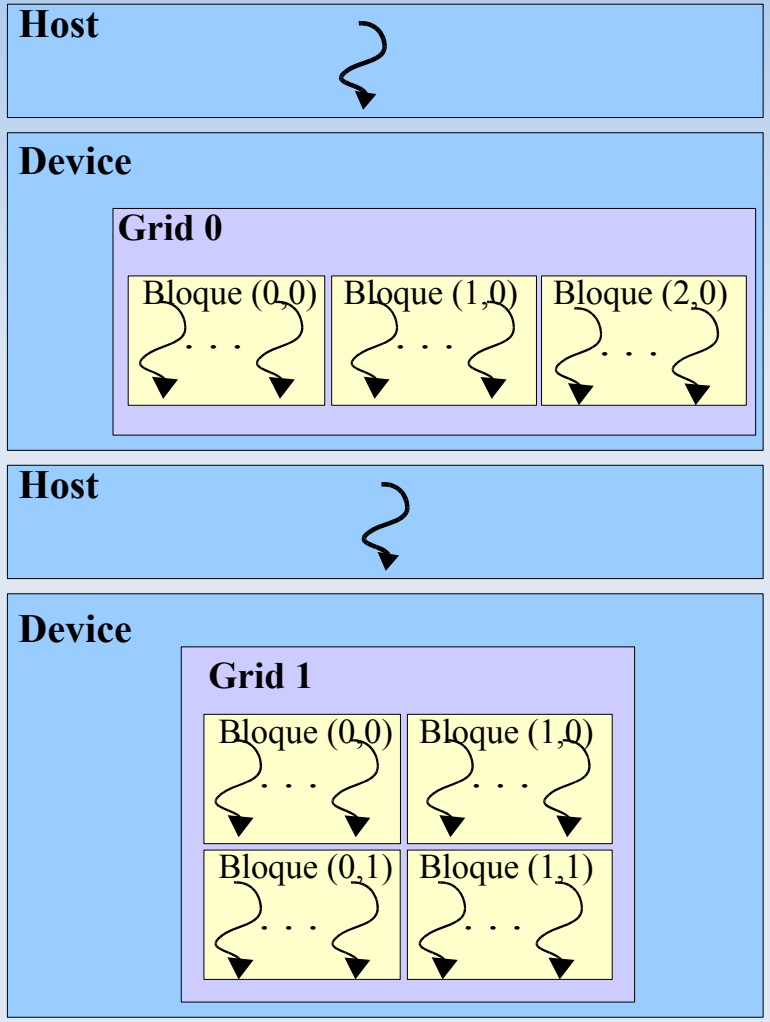
__global__ Kernel1 (...)
{ … }

//Código secuencial

// Kernel paralelo
Kernel0 <<< ... >>> (...)

//Código secuencial

//Kernel paralelo
Kernel1 <<< ... >>> (...)



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

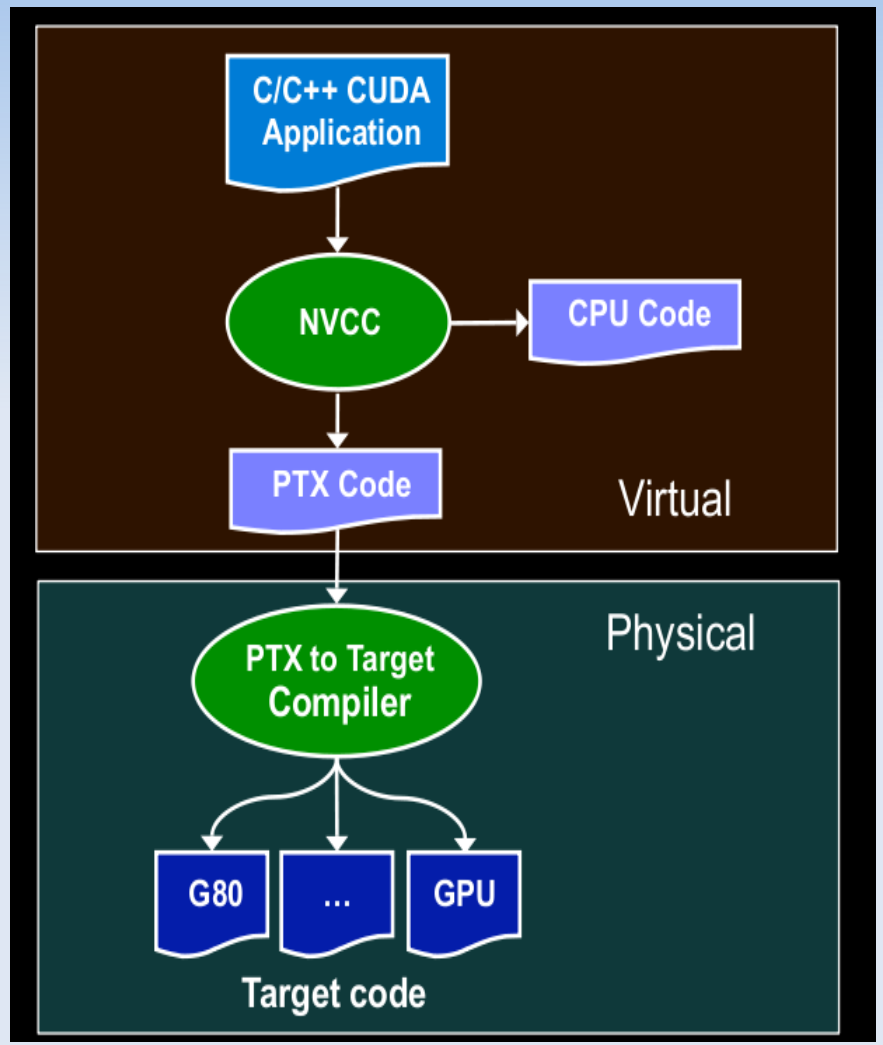
Interfaz de programación
CUDA C

- **Extensión mínima de C** para definir kernels como funciones C +
- API en tiempo de ejecución: implementada con la biblioteca dinámica **cudart**
  - Reservar y liberar memoria en el device
  - Transferir datos entre memoria de host y device
  - Gestionar múltiples devices
- Requiere usar el driver de compilación **nvcc**

- También es posible programar con la API del driver de CUDA pero es de más bajo nivel (más duro).

Compilación código CUDA

- **Dos etapas**
  - Independiente GPU: Genera Código PTX (Parallel Thread eXecution).
  - Física: Genera código objeto para una GPU concreta.
- **Nvcc es el driver de compilación:** Monitoriza llamadas a todos los compiladores y herramientas CUDA: cudac, g++, ...
- **Salida**: Separa código CPU de código GPU.
  - Código C para CPU (debe ser compilado con otra herramienta).
  - Código objeto PTX.
- El ejecutable CUDA usa **dos bibliotecas dinámicas**:
  - Cudart (CUDA runtime)
  - Cuda (CUDA core)



Modificadores

- **Modificadores para funciones a ejecutar en GPU**
  - Función invocada en CPU

    __global__ void Kernel (...) {.....}
  - Función invocada en GPU

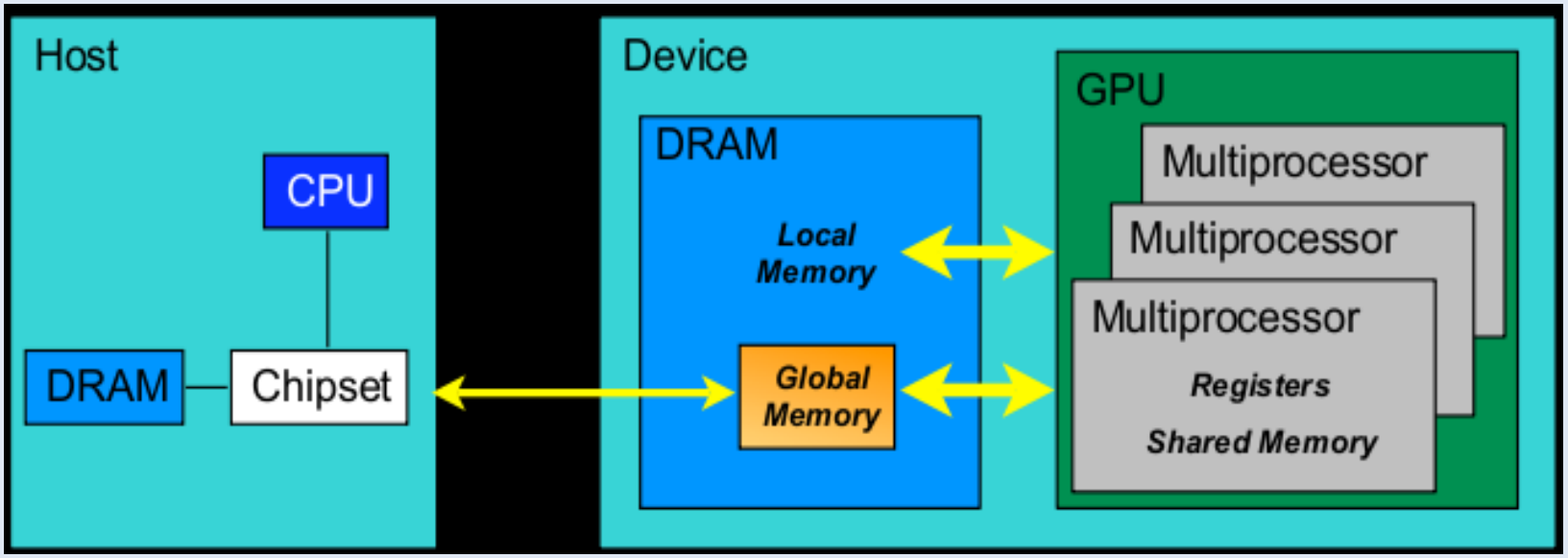
    __device__ int Funcion (...) {.....}
- **Modificadores para variables en device**
  - **Variable en memoria compartida**: **__shared__** float matriz[32];

    - Accesible por todas las hebras dentro del mismo bloque

    - Sólo dura mientras se ejecuta el bloque de hebras
  - **Constante:** **__constant__** float A[64];
  - **Variables sin modificador:** Escalares y vectores de tipos predefinidos se almacenan en registros si caben. En caso contrario, a memoria local.

Gestión memoria device

- CPU y GPU tienen **espacios de memoria separados**.
- Existen funciones de tiempo de ejecución para:
  - Reservar/liberar memoria global device (DRAM)
  - Transferir datos entre memoria global device y memoria host



Reservar/liberar memoria

- Reservar mem. device: **cudaMalloc(void ** puntero, size_t numbytes)**
- Liberar mem. device: **cudaFree(void* puntero)**
- Fijar valor en memoria lineal device

  cudaMemset(void * puntero, int valor, size_t numbytes)

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

Transferencias de datos

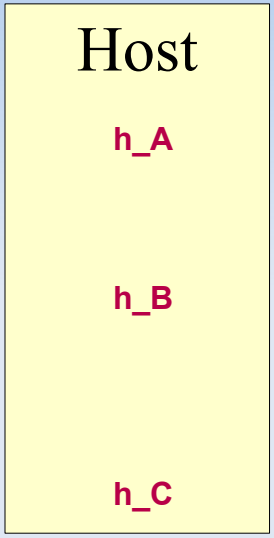
cudaMemcpy(void *destino, void *fuente, size_t nunbytes,

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 a la hebra 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.

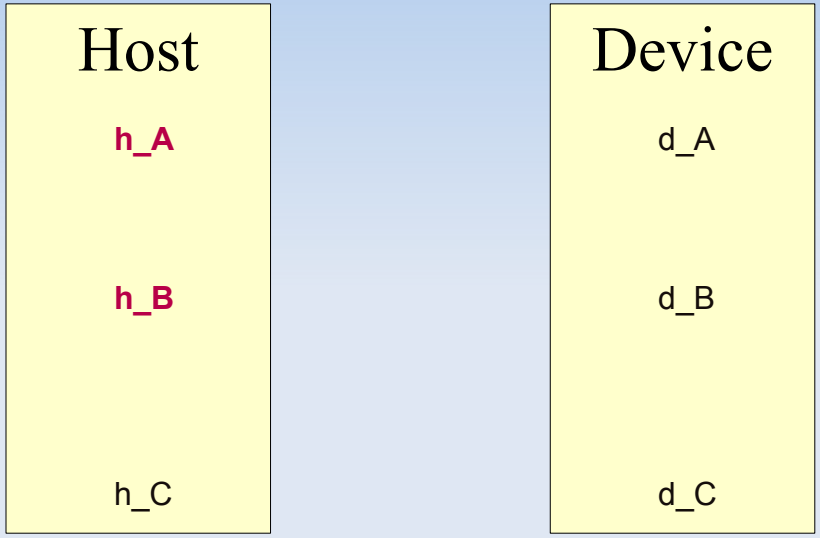
Transferencias. Ejemplo

```
__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 = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid =ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```



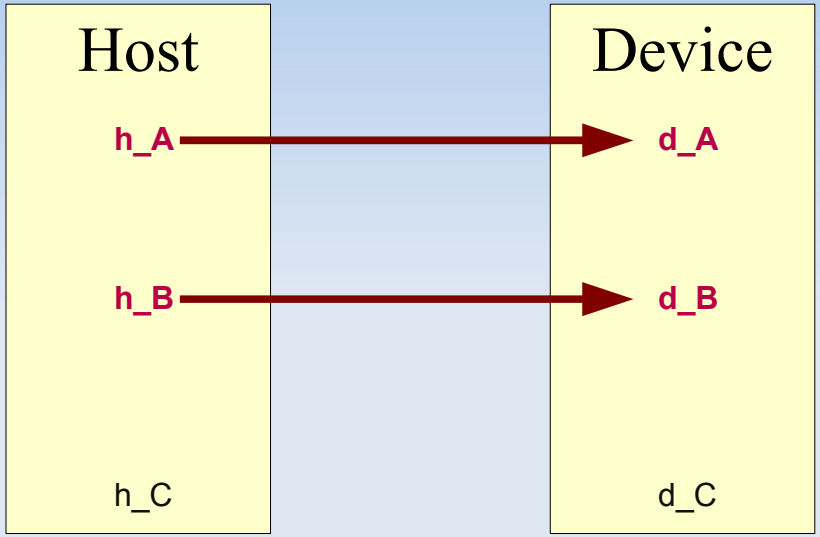
Transferencias. Ejemplo

```
__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 = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid =ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```



Transferencias. Ejemplo

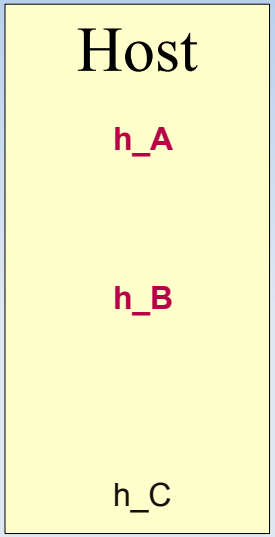
```
__global__ void VecAdd(float* A, float* B, float* C, int N)
    int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)  C[i] = A[i] + B[i]; }

int main()
{ int N = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid =ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```

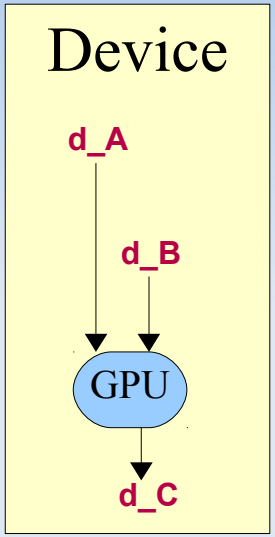


kubuntu

Transferencias. Ejemplo

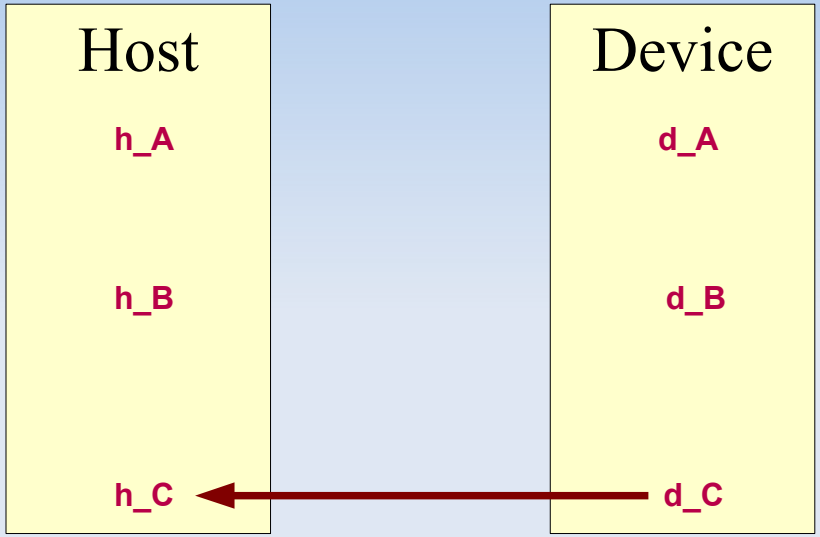
```
__global__ void VecAdd(float* A, float* B, float* C, int N)
    int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)  C[i] = A[i] + B[i]; }

int main()
{ int N = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid=ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```





Transferencias. Ejemplo

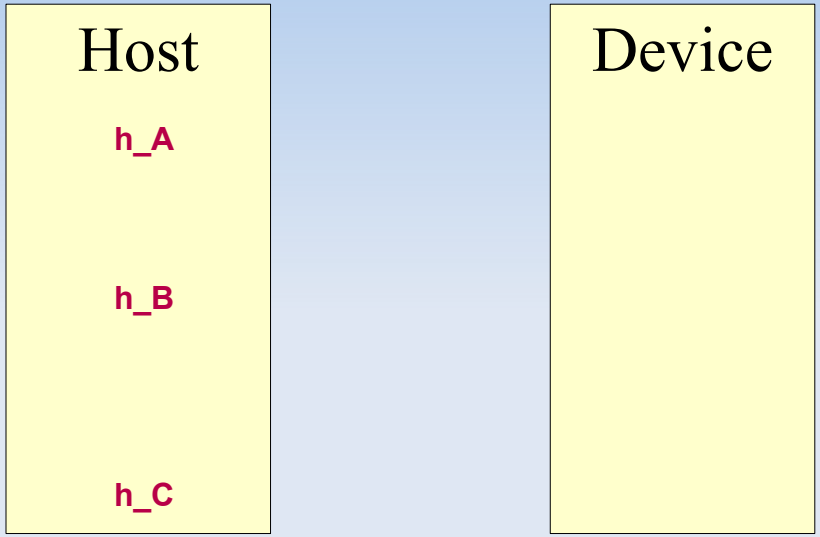
```
__global__ void VecAdd(float* A, float* B, float* C, int N)
    int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)  C[i] = A[i] + B[i]; }

int main()
{ int N = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid=ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```



kubuntu

Transferencias. 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 = ...;   size_t size = N * sizeof(float);
float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
Initialize(h_A,  h_B,  N);
float* d_A;  cudaMalloc((void**)&d_A, size);
float* d_B;  cudaMalloc((void**)&d_B, size);
float* d_C;  cudaMalloc((void**)&d_C, size);
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

int hebrasBloque = 256; int bloquesGrid=ceil(float(N)/hebrasBloque);
VecAdd<<<bloquesGrid, hebrasBloque>>>(d_A, d_B, d_C, N);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A);cudaFree(d_B); cudaFree(d_C); …  }
```



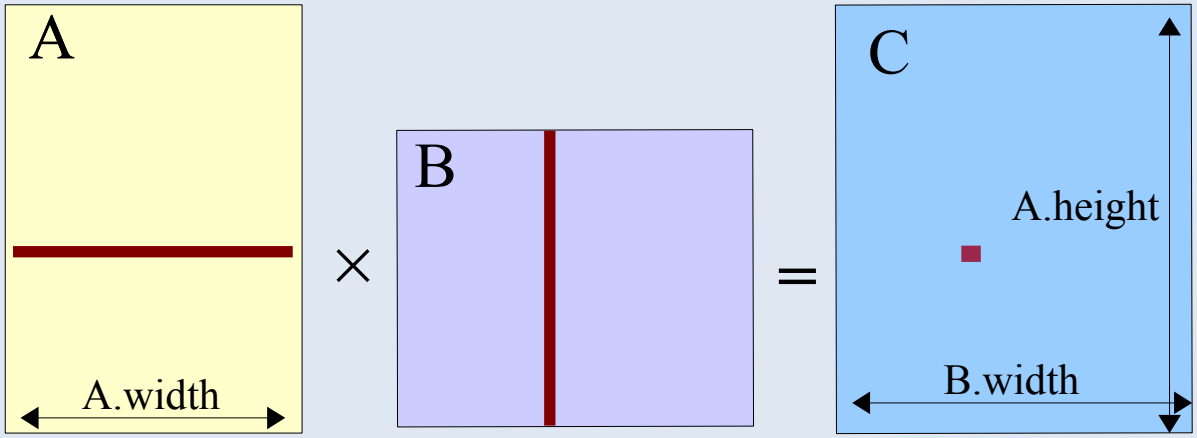
Sincronización de hebras

void __syncthreads();

- **Sincroniza todas las hebras de un mismo bloque**
  - Establece barrera de sincronización entre todas las hebras del bloque
  - Se usa para evitar inconsistencias en el acceso a memoria compartida
- Sólo se puede llamar en **código condicional** si la condición presenta la misma evaluación para todas las hebras del bloque

Uso de la memoria compartida

- **La memoria compartida (16KB-48KB por MP) es muchísimo más rápida que la global.**
- Casi siempre, interesa reemplazar accesos a memoria global con accesos a memoria compartida.
- Puede suponer rediseñar el código para reutilizar datos en memoria compartida.
- **Ejemplo**: Multiplicación matriz-matriz: A × B=C,




```
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
```

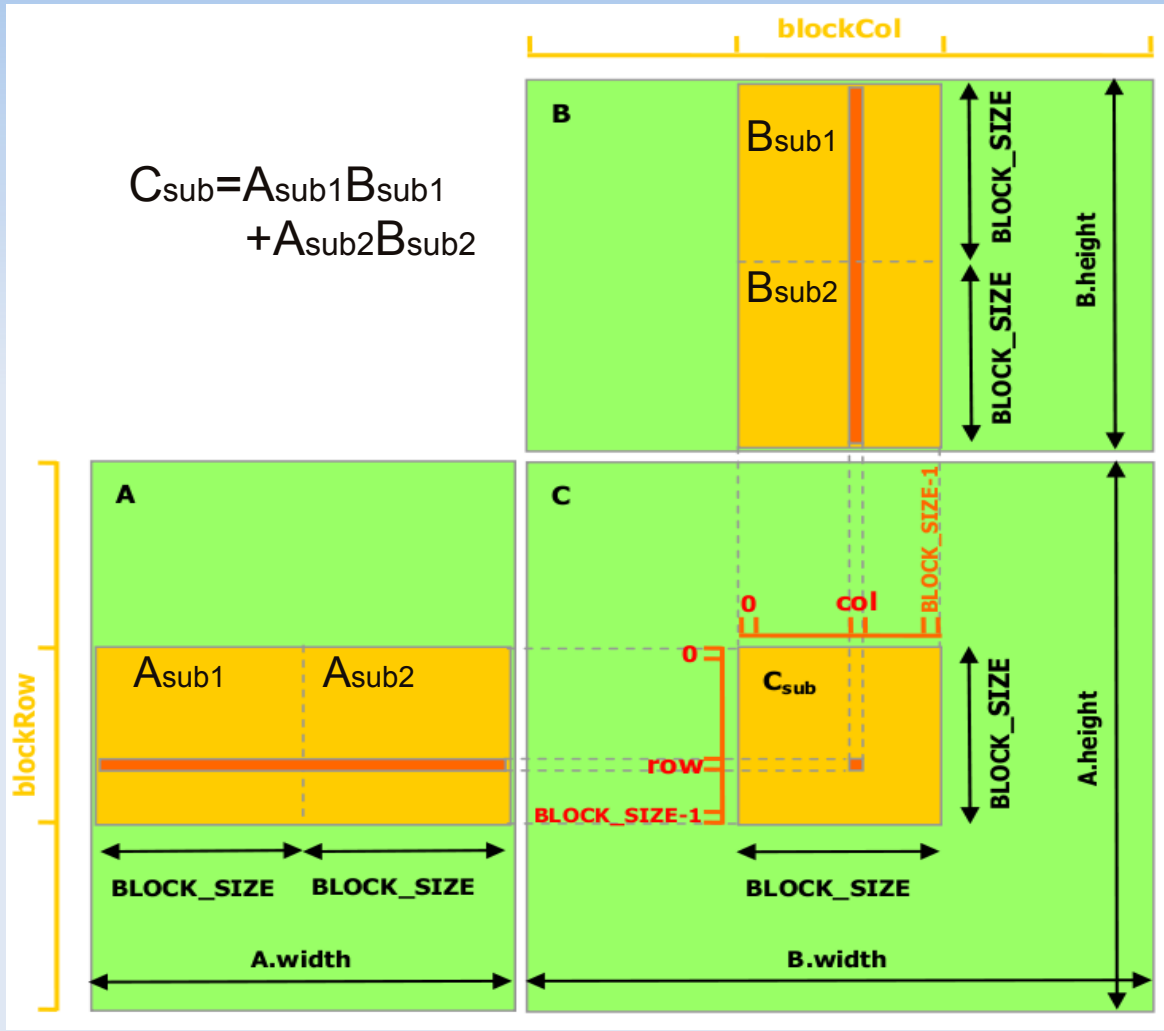
```
M(row, col) = *(M.elements
                + row * M.stride
                + col)
```

Multiplicación matriz-matriz

- Cada bloque de hebras calcula una submatriz cuadrada C_{sub} de C y cada hebra del bloque calcula un elemento de C_{sub}
- Cada C_{sub} es el producto de dos submatrices rectangulares: A_{sub} y B_{sub}.
- A_{sub} y B_{sub} se descomponen en submatrices cuadradas de dimensión BLOCK_SIZE (A_{subi} y B_{subi}) y C_{sub} se calcula como suma de productos de estas submatrices.
- Cada producto se realiza:

  1. Cargando A_{subi} y B_{subi} en mem. compartida (desde mem. Global)

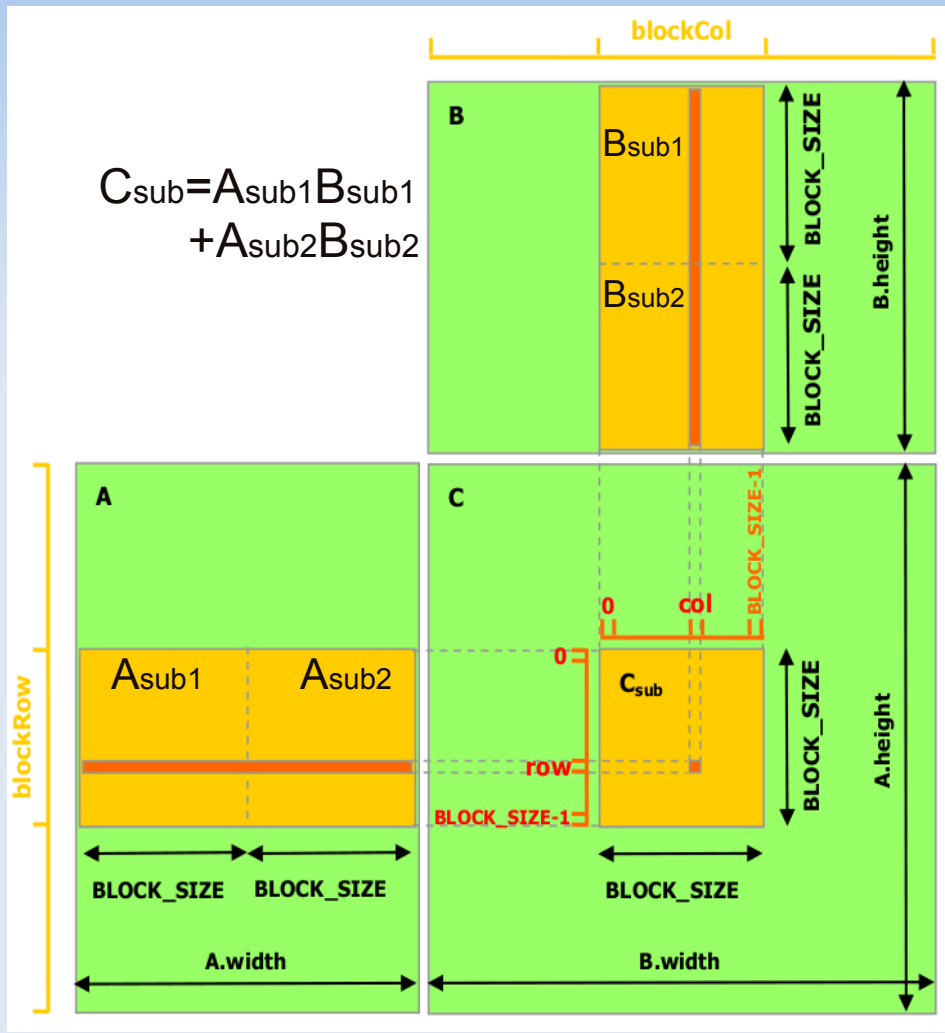
  2. Cada hebra calcula un elemento del producto y guarda resultado en registro parta ir acumulando
- Al finalizar productos cada hebra guarda resultado en mem. global



kubuntu

Multiplicación matriz-matriz
Preliminares

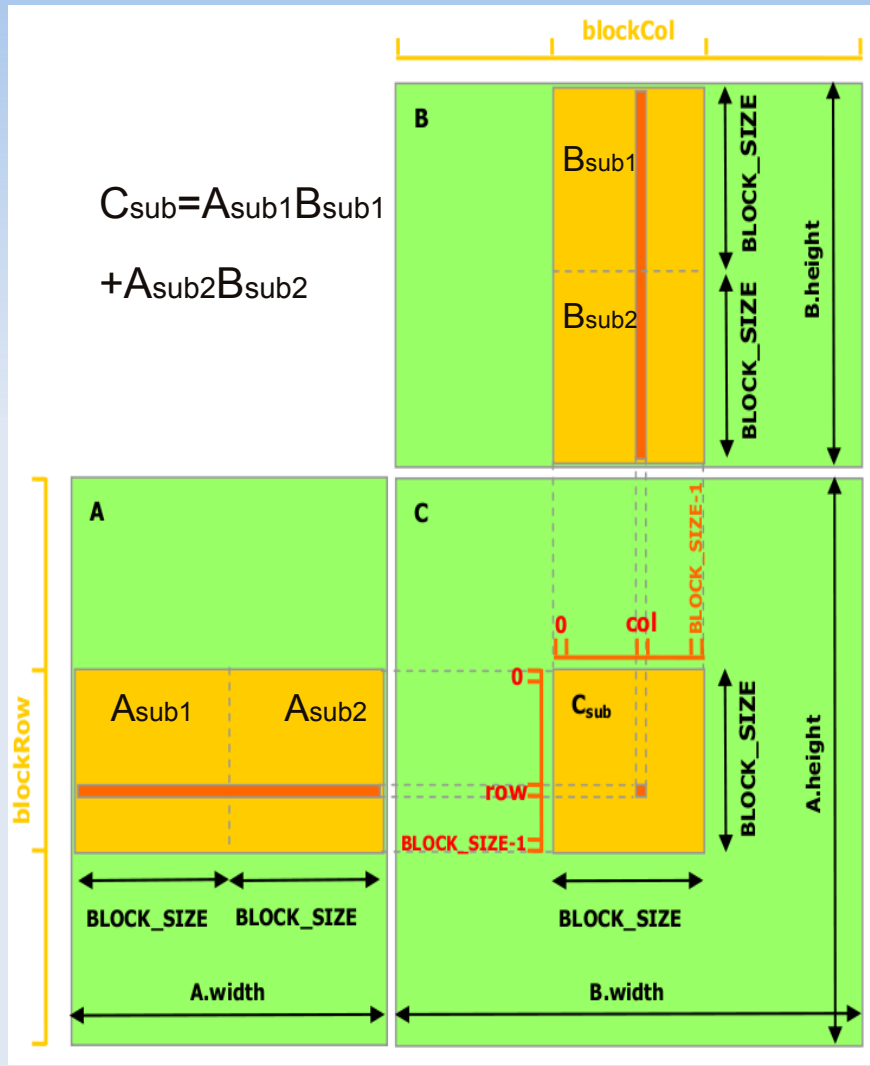
```
#define BLOCK_SIZE 16 // Tamaño del bloque de hebras

// Obtener un elemento de una matriz A
__device__ float GetElement (const Matrix A, int row, int col)
{   return A.elements[row * A.stride + col]; }


// Fijar el valor de un elemento de una matriz A
__device__ void SetElement(Matrix A, int row, int col,float value)
     {  A.elements[row * A.stride + col] = value;}


// Obtener la submatriz Asub que se encuentra col submatrices
   a la derecha y row submatrices abajo del comienzo de A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
   Matrix Asub;
   Asub.width    = BLOCK_SIZE;
   Asub.height   = BLOCK_SIZE;
   Asub.stride   = A.stride;
   Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                        + BLOCK_SIZE * col];
   return Asub;
}
```



kubuntu

Multiplicación matriz-matriz
Función del host

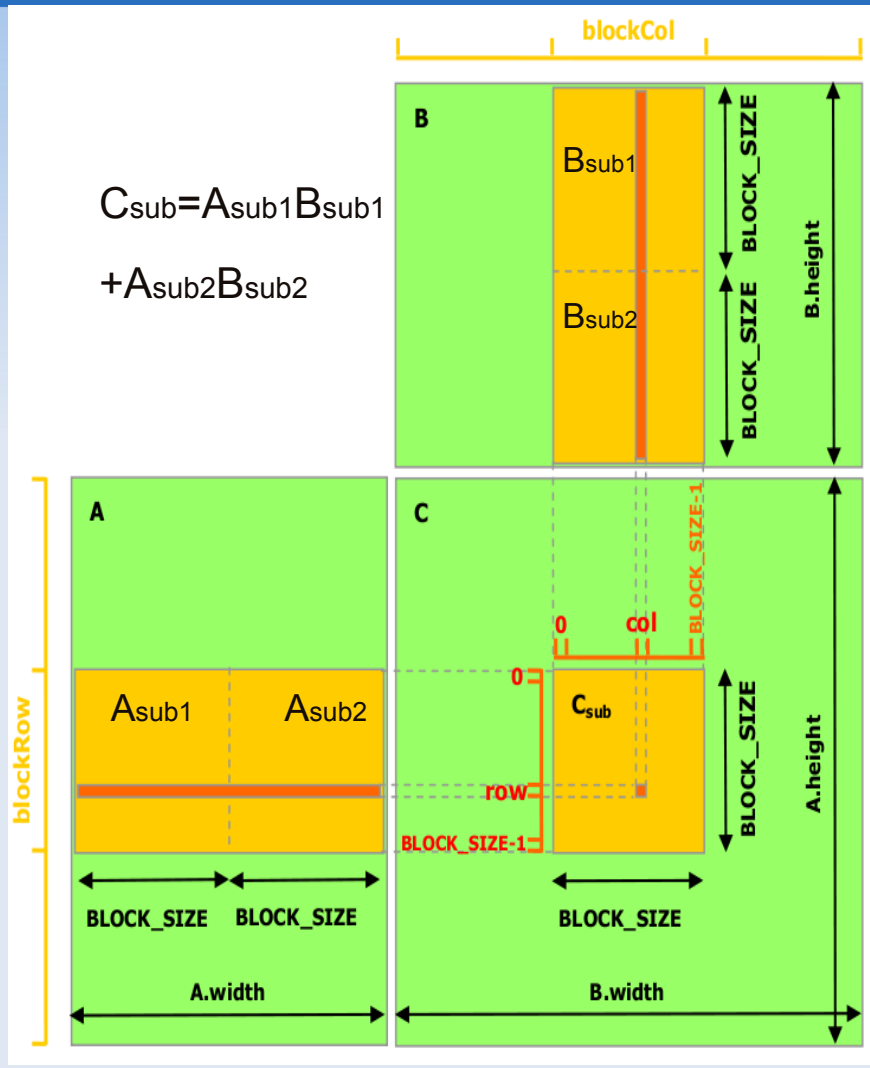
```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
 //Carga A a memoria global device
Matrix d_A; d_A.width = d_A.stride = A.width; d_A.height =
A.height;
 size_t size = A.width * A.height * sizeof(float);
cudaMalloc((void**)&d_A.elements, size);
cudaMemcpy(d_A.elements, A.elements, size,
cudaMemcpyHostToDevice);
//Carga B a memoria global device
      .....
  // Reserva memoria para C en device
Matrix d_C;d_C.width = d_C.stride = C.width; d_C.height = C.height;
 size = C.width * C.height * sizeof(float);
 cudaMalloc((void**)&d_C.elements, size);
  // Llamada al  kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
  MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

// Lee C de device
  cudaMemcpy(C.elements, d_C.elements, size,
                                        cudaMemcpyDeviceToHost);
   // Libera memoria device ...}
```



kubuntu

Multiplicación matriz-matriz Kernel

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
   int blockRow = blockIdx.y;    int blockCol = blockIdx.x;
   Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
   float Cvalue = 0; // Variable para guardar resultado
   int row = threadIdx.y;  int col = threadIdx.x;
   //Bucle para multiplicar submatrices  Asubi y Bsubi
   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

     Matrix Asub = GetSubMatrix(A, blockRow, m);  // Obten Asub de A
     Matrix Bsub = GetSubMatrix(B, m, blockCol);    // Obten Bsub de B
     // Declara y carga variables en memoria compartida
     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
                    As[row][col] = GetElement(Asub, row, col);
                    Bs[row][col] = GetElement(Bsub, row, col);

     __syncthreads(); // Sincroniza para asegurar carga
     // Multiplica Asubi y Bsubi para actualizar Cvalue
     for (int e = 0; e < BLOCK_SIZE; ++e)
                 Cvalue += As[row][e] * Bs[e][col];
     __syncthreads(); // Sincroniza para asegurar fin cómputo previo }

  SetElement(Csub, row, col, Cvalue);   // Escribe Csub a memoria global
}
```



kubuntu