CUDA: Introducción y algunos tópicos avanzados

Ursula Iturrarán-Viveros

Facultad de Ciencias U.N.A.M.

Seminario de Tecnologías de Alto Desempeño Aplicadas a la Modelación Matemática y Computacional

Instituto de Geofísica U.N.A.M.

14-Abril-2011

Contenido

- Introducción
- Modelo de programación (conceptos básicos y tipos de datos)
- Modelo de memoria
- CUDA básicos
- Ejemplo simple para una función kernel

CUDA

- “Compute Unified Device Architecture”
- Modelo de programación de propósito general
  - El usuario manjea los lotes de threads (hilos) en el GPU
  - GPU = co-procesador masivamente paralelo, dedicado a proceso de threads
- Librerias disponibles
  - CUDA SDK código muestra
  - CUDA BLAS
  - CUDA FFT
  - Rutinas para matrices sparse TRUST y CUSP

Cómputo paralelo con GPUs

- 8-series de GPUs dan 25 to 200+ GFLOPS en aplicaciones compiladas en C paralelo
  - Disponible en laptops, desktops y clusters
- El paralelismo de los GPUs se duplica cada año
- Los modelos de programación escalan transparentemente
- GPUs Programables en C con herramientas de CUDA

Modelo de programación con CUDA

- El GPU se puede ver como un *dispositivo de cómputo* para ejecutar una porción de una aplicación que:
  - Tiene que ejecutarse muchas veces
  - Puede aislarse como función
  - Trabaja independientemente con datos diferentes
  - Tiene su propia memoria DRAM (device memory)
- Tal función puede compilarse para correr en el dispositivo (*device*). El programa resultante se llama un Kernel

Palabras clave

- Kernel
  - Código que puede correr en los procesadores del GPU
- Thread (hilos de ejecución)
  - Una ejecución de un kernel con un índice dado. Cada thread usa su índice para tener acceso a los elementos en un arreglo. La colección de todos los threads cooperan para procesar todo el conjunto de datos.

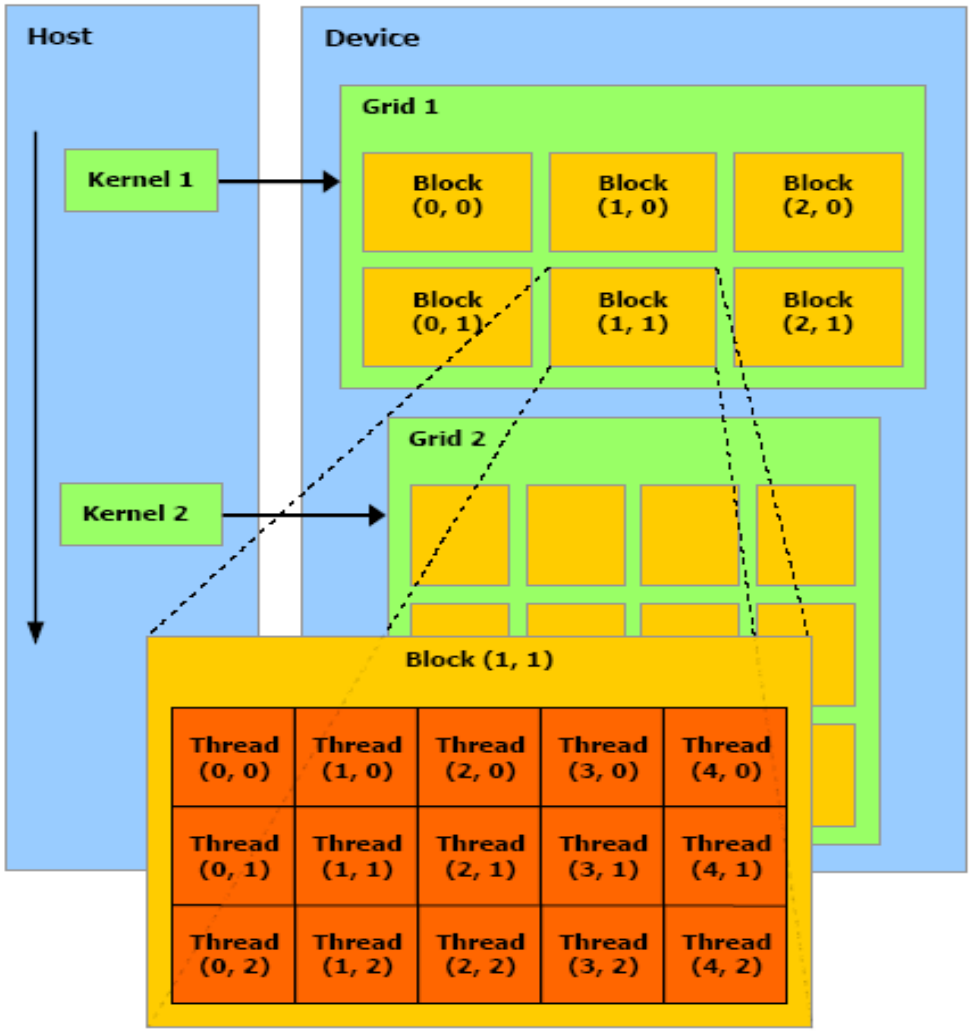
Modelo de Programación

- Bloque de Threads
  - Conjunto de threads que pueden cooperar
    - Memoria compartida rápida
    - Se pueden sincronizar: _syncthreads() función que actua como una barrera
    - ID para cada Thread
  - El Bloque puede ser un arreglo de una-, dos- o tres-dimensiones
- Grid
  - Es un grupo de bloques(thread).
  - No hay sincronización entre los bloques.

Modelo de Programación

- Grid de Bloques de Threads
  - Número limitado de threads en un bloque
  - Permite numeros grandes de threads para ejecutar el mismo kernel con una sola llamada
  - Bloque identificables via un identificador de bloque: ID

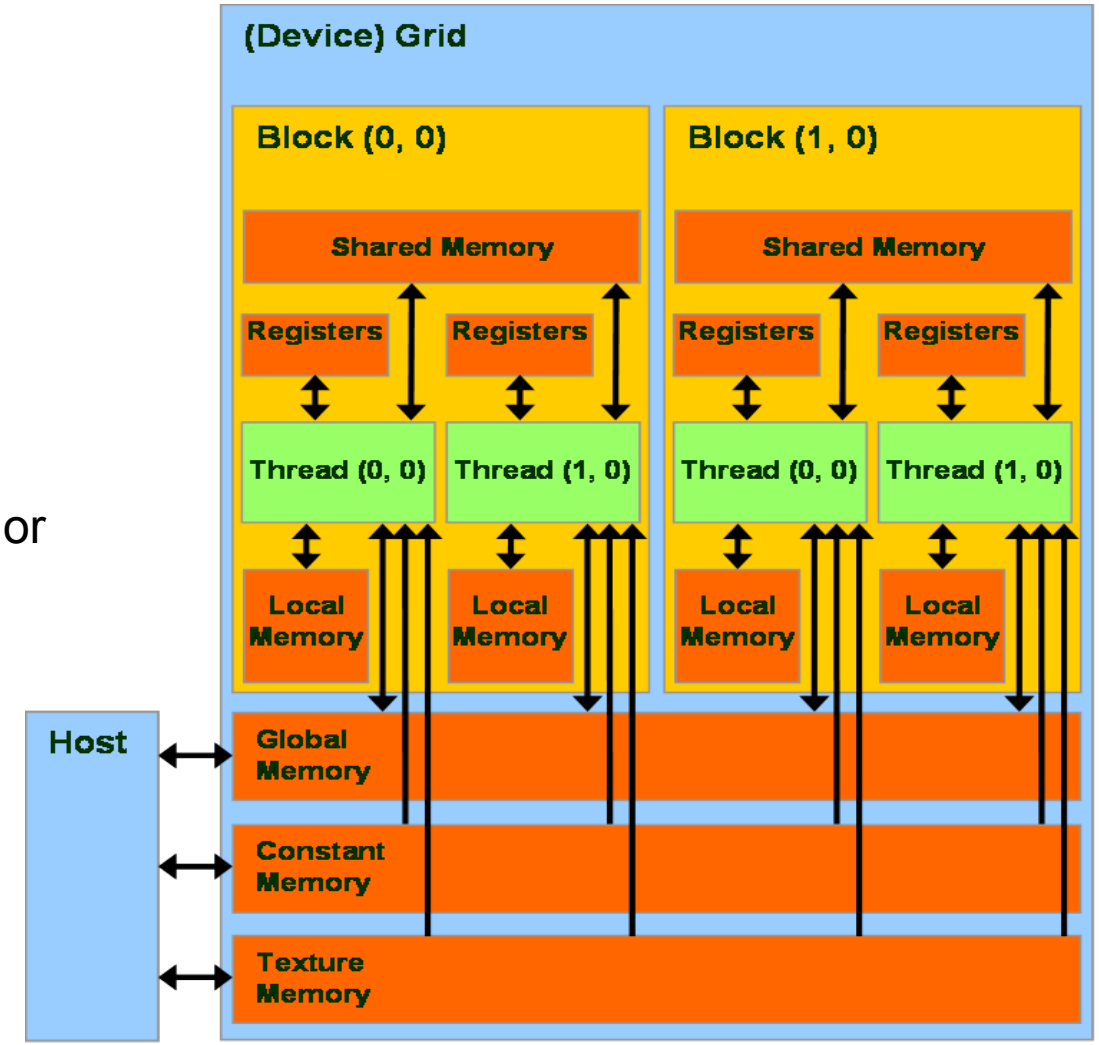
CUDA Programming Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Modelo de memoria con CUDA

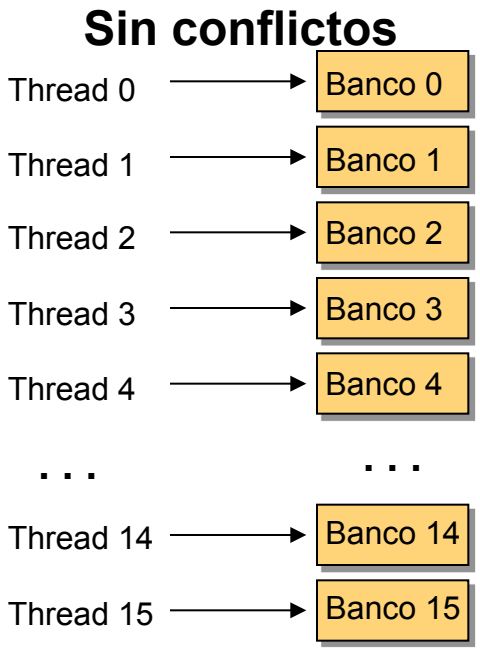
- **`cudaMalloc()`**
  - Obtiene espacio en la memoria global
  - Parámetros: dirección del apuntador y el tamaño a reservar
- **`cudaMemset()`**
  - Inicializa a un valor dado
  - Parámetros: dirección, valor y cantidad
- **`cudaFree()`**
  - Libera el espacio
  - Parámetros: dirección



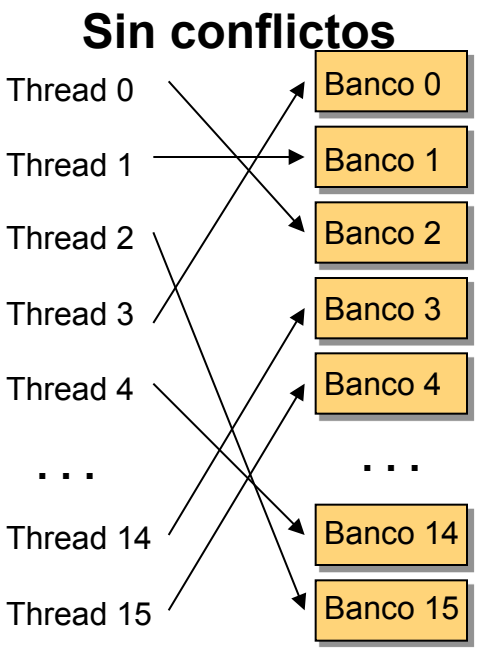
Modelo de memoria en CUDA

- Memoria compartida
  - Esta en el-chip:
    - Mucho más rápida que la memoria local o global,
    - Tan rápida como un registro cuando no hay conflictos de bancos de datos,
    - Dividida en modulos o bancos de datos de igual tamaño
  - Palabras sucesivas de 32-bit son asignadas a bancos sucesivos,
  - Cada banco tiene un ancho de banda de 32 bits por ciclo de reloj.

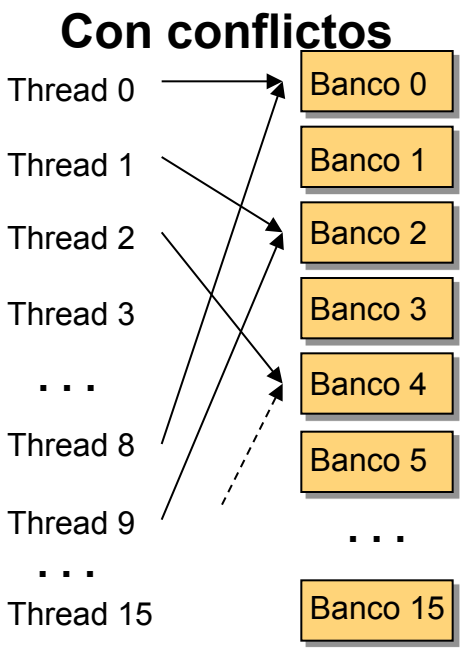
Espacios de memoria en CUDA



Direccionamiento lineal
Stride = 1



Direccionamiento aleatorio
Permutación 1:1



Direccionamiento lineal
Stride = 2

12

C extendido

- **Especificación de declaraciones**
  - **global, device, shared, local, constant**
- **Palabras clave**
  - **threadIdx, blockIdx**
- **Intrinsecos**
  - **__syncthreads**
- **Runtime API**
  - **Memoria, ejecución administración**
- **Llamada de Función**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Básicos

- Una extensión al lenguaje de programación de C
  - Función tipo, calificadores para especificar la ejecución en el host o en el dispositivo
  - Variable tipo, calificadores para especificar la localidad de memoria en el dispositivo
  - Una nueva directiva para especifcar como ejecutar un kernel en el dispositivo
  - Cuatro variables que especifican el tamaño del grid y bloques y los indices de los threads

CUDA Básicos

- Calificadores del tipo de Function

__device__

- Ejecutado en el dispositivo
- Solo se puede llamar desde el dispositivo.

__global__

- Ejecutado en el dispositivo,
- Solo se puede llamar en el host.

__host__

- Ejecutado en el host,
- Solo se puede llamar en el host.

CUDA Básicos

- Calificadores del tipo de Variable

__device__
  - Reside en el espacio de memoria global,
  - Tiene el ciclo de vida de una aplicación,
  - Es accesible desde todos los threads dentro de un grid y desde el host a través de la librería runtime.

__constant__ (opcional usado junto con **__device__**)
  - Reside en un espacio constante de memoria,
  - Tiene el ciclo de vida de una aplicación,
  - Es accesible desde todos los threads dentro de un grid y desde el host a través de la librería runtime. .

__shared__ (opcional usado junto con **__device__**)
  - Reside en un espacio compartido de memoria de un thread o un bloque,
  - Tiene el ciclo de vida de un bloque,
  - Es accesible unicamente desde todos los threads dentro de un bloque.

CUDA Básicos

- Configuración de Ejecución
  - Deben de especificarse para cualquier llamada a una función **__global__ :**
  - Definir la dimensión del grid y de los bloques
  - Esto se especifica insertando una expresión entre el nombre de la función y la lista de argumentos:

    función:

    ```
    __global__ void Func(float* parameter);
    ```

    debe de llamarse como:

    ```
    Func<<< Dg, Db, Ns >>>(parameter);
    ```

CUDA Básicos

- Donde `Dg, Db, Ns` son :
  - **Dg** es del tipo **dim3** → dimension y tamaño del grid
    - **Dg.x** * **Dg.y** = número de bloques que se lanzan;
  - **Db e**s del tipo **dim3** → dimensión y tamaño de cada bloque
    - **Db.x** * **Db.y** * **Db.z** = número de threads por bloque;
  - **Ns e**s del tipo **size_t** → número de bytes en la memoria compartida
    - **Ns** es un argumento opcional, el dafault es 0.

CUDA Básicos

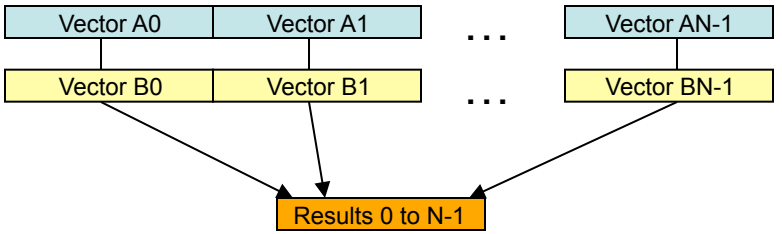
- Variables
  - **`gridDim`** es del tipo **dim3** →dimensiones del grid.
  - **`blockIdx`** es del tipo **uint3** → índice del bloque dentro del grid.
  - **`blockDim`** es del tipo **dim3** → dimensiones del bloque.
  - **`threadIdx`** es del tipo **uint3** → índice del thread dentro del bloque.

Ejemplo: Producto escalar

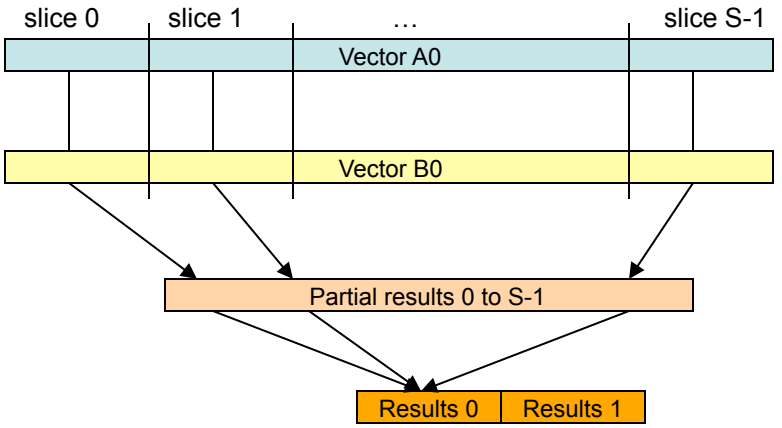
- Calcular el producto escalar de
  - 32 pares de vectores
  - 4096 elementos en cada uno
- Una forma eficiente para correr en el dispositivo es organizar el cálculo en:
  - Un grid de 32 bloques
  - Con 256 threads por bloque
- Esto nos 4096/265 = 16 rebanadas por vector

Ejemplo: Producto escalar

- Los datos se entregan al dispositivo como dos arreglos y los resultados serán almacenados en un arreglo de resultados



- Cada producto de un par de vectores An, Bn será calculado en rebanadas que serán agregadas hasta obtener el resultado final.



Ejemplo: Producto escalar

El programa en el host

```
int main(int argc, char *argv[]){

  CUT_CHECK_DEVICE();
  …

  h_A = (float *)malloc(DATA_SZ);
  …

  cudaMalloc((void **)&d_A, DATA_SZ);
  …

  cudaMemcpy(d_A, h_A, DATA_SZ, cudaMemcpyHostToDevice);
  …

  ProdGPU<<<BLOCK_N, THREAD_N>>>(d_C, d_A, d_B);
  …

  cudaMemcpy(h_C_GPU, d_C, RESULT_SZ,
    cudaMemcpyDeviceToHost);
  …

  CUDA_SAFE_CALL( cudaFree(d_A)   );
  free(h_A);
  …

  CUT_EXIT(argc, argv);

}
```

Ejemplo: Producto escalar

La función Kernel

- Parámetros:
  - d_C: apuntador al arreglo de resultados
  - d_A, d_B apuntadores a los datos de entrada
- Arreglos locales de datos:
  - t[]: resultados de un solo thread
  - r[]: fracción de cache
- I: ID del thread en el bloque

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){
  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;

  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x)
    {
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
    THREAD_N){
      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();

     for(int stride = THREAD_N / 2; stride > 0; stride /= 2){
        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }

      if(I == 0) r[slice] = t[0];

    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){

      if(I < stride) r[I] += r[stride + I];
      __syncthreads();

    }
    if(I == 0) d_C[vec_n] = r[0];

  }
}
```

Ejemplo: Producto escalar

La función Kernel

- Corre a través de cada par de vectores de entrada
- Para nuestros numeros solo será ejecutado una vez pues:
  Dimensión del Grid == número de vectores

→ número de vector = ID del bloque

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){

  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;

  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x){
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
    THREAD_N){

      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();

      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();

      }

      if(I == 0) r[slice] = t[0];

    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){

      if(I < stride) r[I] += r[stride + I];
      __syncthreads();

    }
    if(I == 0) d_C[vec_n] = r[0];
  }
}
```

Ejemplo: Producto escalar

La función Kernel

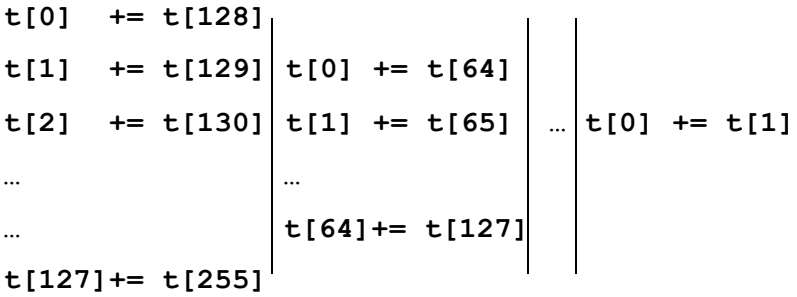
- Corre a través de cada sección o rebanada de vectores de entrada
- Cada thread calcula un solo producto y lo salva

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){

  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;

  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x){
    int base = ELEMENT_N * vec_n;

    for(int slice = 0; slice < SLICE_N; slice++, base += THREAD_N)
    {

      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();

      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();
      }

      if(I == 0) r[slice] = t[0];

    }

    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){

      if(I < stride) r[I] += r[stride + I];
      __syncthreads();

    }
    if(I == 0) d_C[vec_n] = r[0];
  }
}
```

Ejemplo: Producto escalar

La función Kernel

- Calcula el resultado parcial para cada rebanada



- Salva el resultado parcial

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){

  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;

  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x){
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
    THREAD_N){

      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();

      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();

      }

      if(I == 0) r[slice] = t[0];

    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){

      if(I < stride) r[I] += r[stride + I];
      __syncthreads();

    }
    if(I == 0) d_C[vec_n] = r[0];
  }
}
```

Ejemplo: Producto escalar

La función Kernel

- Suma los resultados para todas las rebanadas
- Salva el resultado en la memoria del dispositivo

```
__global__ void ProdGPU(float *d_C, float *d_A, float *d_B){

  __shared__ float t[THREAD_N];
  __shared__ float r[SLICE_N];
  const int I = threadIdx.x;

  for(int vec_n=blockIdx.x; vec_n<VECTOR_N; vec_n+=gridDim.x){
    int base = ELEMENT_N * vec_n;
    for(int slice = 0; slice < SLICE_N; slice++, base +=
    THREAD_N){

      t[I] = d_A[base + I] * d_B[base + I];
      __syncthreads();

      for(int stride = THREAD_N / 2; stride > 0; stride /= 2){

        if(I < stride) t[I] += t[stride + I];
          __syncthreads();

      }

      if(I == 0) r[slice] = t[0];

    }
    for(int stride = SLICE_N / 2; stride > 0; stride /= 2){

      if(I < stride) r[I] += r[stride + I];
      __syncthreads();

    }
    if(I == 0) d_C[vec_n] = r[0];
  }
}
```


GRACIAS
Yupi Internet Inc.©