Introducción a la Programación en Tarjetas de Video

CUDA

Edison Montoya

Centro de Radioastronomía y Astrofísica

UNAM

Marzo 28 de 2014

CUDA

Se refiere a dos cosas:

- Arquitectura en la que se basan las tarjetas de video de Nvidia.

- Lenguaje de programación con el cual se puede controlar la tarjeta de video (esta charla). CUDA se puede acoplar con diferentes lenguajes de programación:
    - Mathematica
    - IDL
    - Java
    - Matlab
    - Python
    - Fortran
    - C

Nosostros nos restingiremos a CUDA C.

Prerrequisitos y Terminología

Para sacar provecho de la charla se necesita:

- Conocimientos básicos en C ó Fortran
- Manejo de punteros

No se necesita saber como funciona una tarjeta de video o experiencia en paralelizar códigos.

Algunos términos que usaremos son:

Host → Es la CPU y su memoria.

Device → Es la tarjeta de video, compuesta de una GPU (graphics processing unit) y su memoria.

Por qué programar para GPU?

- Costos: 10 (o más) veces más económico que su equivalente en CPU.
- Energía: consume la mitad de energía.
- Fácil acceso: todas las computadoras actuales tienen tarjeta de video y los códigos no estan restringidos a correr en tarjetas Nvidia. Para escribir códigos que corrar en “cualquier” tarjeta de video se puede usar OpenCL que tiene una lógica de programación similar a la de CUDA C.

Motivaciones Extras

Capacidad de Procesamiento

Una sola tarjeta de video puede llegar a realizar más cantidad de operaciones por segundo que 50 CPUs.

Ejemplo:

- Consideremos la tarjeta de video GeForce GT 425M (96 CUDA cores, 1GB memoria) de una laptop, la cual ha mostrado en la práctica hacer el mismo número FLOPS (operaciones de punto flotante por segundo) que 8 CPUs (2 procesadores Xeon E5420, 2.5 GHz).
- Otro ejemplo sería la tarjeta Quadro 4000 (256 CUDA cores, 2GB memoria) que ha puesto a disposición la *Coordinación de Investigación Científica (UMSNH)*, la cual ha mostrado en la práctica una capacidad equivalente a 18 CPUs (~5 procesadores Xeon E5420, 2.5 GHz).

Nota: La medida de la capacidad de estas tarjetas puede variar, ya que los valores aquí mencionados son para programas que no están optimizados.

Hola Mundo!!

Simple

```c
#include <stdio.h>

int main(void){

  printf("Hola, soy tu esclavo!\n");

  return 0;
}
```

Condimentado

```c
#include <stdio.h>

__global__ void kernel(){
}

int main(void){

  kernel <<<1,1>>> ();

  printf("Hola, soy tu esclavo!\n");

  return 0;
}
```

El código de la derecha contiene dos elementos adicionales, que aunque no son necesarios, nos ayudan a ilustrar el uso del kernel.

Kernel

Los dos elementos adicionales son:

- __global__, esto dice de que tipo es el kernel:
  - __global__ , se ejecuta en la tarjeta y se llama desde la CPU.
  - __device__, se ejecuta en la tarjeta y se llama desde la tarjeta.
  - __host__ , se ejecuta en la CPU y se llama desde la CPU.

- Kernel <<<1,1>>> ();

  Es un llamado a la función kernel, es similar a los llamados en C, solo que contiene un elemento extra: <<<1,1>>> del que hablaremos luego.

Como Asignar un Valor en la GPU

```c
#include <stdio.h>

// Main program
int main(void){

  // Host memory
  int *a;

  // size of integer
  int size = sizeof(int);

  // Allocate host memory
  a = (int *) malloc(size);

  // Assign value
  *a = 1;

  // Print result
  printf("%d\n",*a);

  // Free host memory
  free(a);

  return 0;
}
```

```c
#include <stdio.h>

// Kernel to give a value
__global__ void value( int *a ){
  *a = 1;
}

// Main program
int main(void){
  int *a;                     // Host memory
  int *a_dev;                 // Device memory
  int size = sizeof(int);     // size of integer

  a = (int *) malloc(size);           // Allocate host memory

  cudaMalloc( (void**) &a_dev, size);  // Allocate device memory

  value <<<1,1>>> (a_dev);            // Launch kernel on device

  // Copy device result back to host
  cudaMemcpy( a, a_dev, size, cudaMemcpyDeviceToHost );

  printf("%d\n",*a);   // Print result

  cudaFree(a_dev);     // Free device memory

  free(a);             // Free host memory

  return 0;
}
```

Tres Elementos Nuevos

cudaMalloc((void**) &a_dev, size);

Asigna una memoria en la tarjeta de tamaño size al puntero a_dev .
Análogo a la función malloc de C.

cudaMemcpy(a, a_dev, size, cudaMemcpyDeviceToHost);

Copia lo que esta almacenado en la memoria de la tarjeta a la memoria de la CPU (cudaMemcpyDeviceToHost) o de la memoria de la CPU a la memoria de la tarjeta (cudaMemcpyHostToDevice)

cudaFree(a_dev);

Libera la memoria, tal como lo hace la función free() en C.

Suma

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

int main(void){

  int a,b,c;

  // Initialize
  a = 1; b = 2;

  // Sum
  c = a + b;

  return 0;
}
```

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

int main(void){

  int *a,*b,*c;
  int size = sizeof(int);

  // Allocate host memory
  a = (int *) malloc (size);
  b = (int *) malloc (size);
  c = (int *) malloc (size);

  // Initialize
  *a = 1; *b = 2;

  // Sum
  *c = *a + *b;

  // Free host memory
  free(a);
  free(b);
  free(c);

  return 0;
}
```

Suma en la Tarjeta de Video (parte 1)

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

// Kernel to add two integers
__global__ void add(int *a, int *b, int *c){
  *c = *a + *b;
}

// Main program
int main(void){

  int *a,*b,*c;               // Host copies
  int *a_dev,*b_dev,*c_dev;  // Device copies
  int size = sizeof(int);

  // Allocate host memory
  a = (int *) malloc (size);
  b = (int *) malloc (size);
  c = (int *) malloc (size);

  // Allocate device memory
  cudaMalloc( (void**)&a_dev, size);
  cudaMalloc( (void**)&b_dev, size);
  cudaMalloc( (void**)&c_dev, size);

  // Initialize
  *a = 1;
  *b = 2;
```

Suma en la Tarjeta de Video (parte 2)

```
    // Copy inputs to device
    cudaMemcpy( a_dev, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( b_dev, b, size, cudaMemcpyHostToDevice );

    // Launch kernel on device
    add <<<1,1>>> (a_dev,b_dev,c_dev);

    // Copy device result back to host
    cudaMemcpy( c, c_dev, size, cudaMemcpyDeviceToHost );

    // Print result
    printf("%d\n",*c);

    // Free device memory
    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Suma en Paralelo en la Tarjeta de Video (parte 1)

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

#define N 128

// Kernel to add N integers
__global__ void add(int *a, int *b, int *c){
  c[????] = a[????] + b[????];
}

// Main program
int main(void){
  int *a,*b,*c;               // Host copies
  int *a_dev,*b_dev,*c_dev;   // Device copies
  int size = N*sizeof(int);   // Size of N integers

  // Allocate host memory
  a = (int *) malloc (size);
  b = (int *) malloc (size);
  c = (int *) malloc (size);

  // Allocate device memory
  cudaMalloc( (void**)&a_dev, size);
  cudaMalloc( (void**)&b_dev, size);
  cudaMalloc( (void**)&c_dev, size);

  // Initialize
  for (int i=0; i<N; i++){
    a[i] = i;
    b[i] = i;
  }
```

Notese los signos de interrogación. Esto es lo que se debe cambiar para Paralelizar.

El tamaño cambia.

Suma en Paralelo en la Tarjeta de Video (parte 2)

```
    // Copy inputs to device
    cudaMemcpy( a_dev, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( b_dev, b, size, cudaMemcpyHostToDevice );

    // Launch kernel on device
    add <<< ?,? >>> (a_dev,b_dev,c_dev);

    // Copy device result back to host
    cudaMemcpy( c, c_dev, size, cudaMemcpyDeviceToHost );

    // Print result
    for (int i=0; i<N; i++)  printf("%d\n",c[i]);

    // Free device memory
    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Notese los signos de interrogación. Esto es lo que se debe cambiar para Paralelizar.

Kernel Paralelizado

Para paralelizar hay dos formas:

1) Usando Bloques

```
// Kernel to add N integers with N parallel blocks
__global__ void add(int *a, int *b, int *c){
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

// Launch kernel on device
add <<<N,1>>> (a_dev,b_dev,c_dev);
```

2) Usando Hilos

```
// Kernel to add N integers with N parallel threads
__global__ void add(int *a, int *b, int *c){
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

// Launch kernel on device
add <<<1,N>>> (a_dev,b_dev,c_dev);
```

Cual es la diferencia entre usar hilos o bloques?

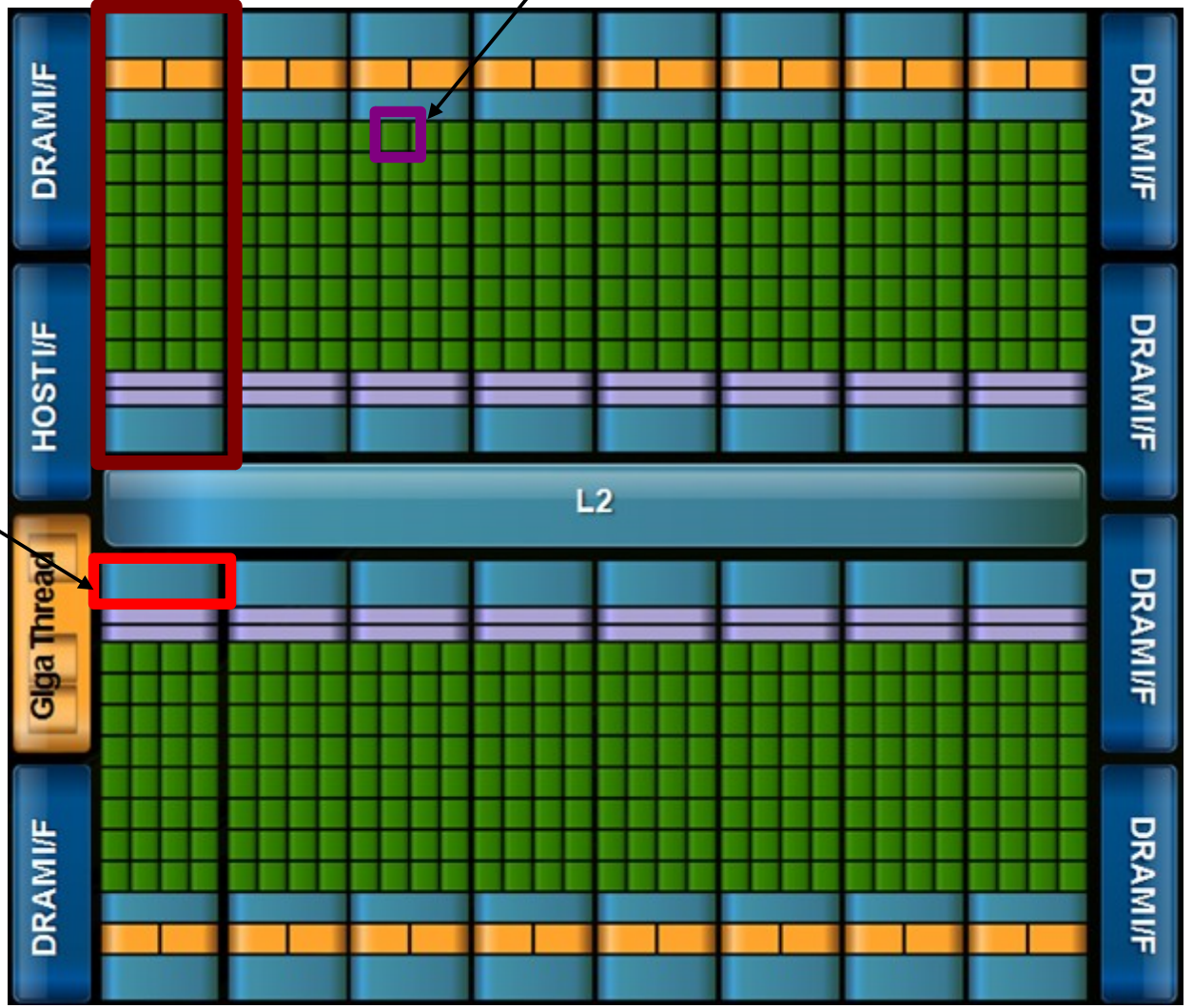
Para responder esto hay que saber algo de la estructura interna de la tarjeta.

Estructura Interna de la Tarjeta de Video

Multiprocesador: ejecuta los bloques

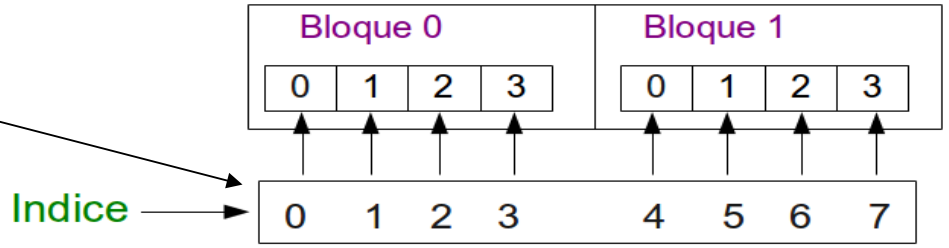
CUDA Core: ejecuta los hilos

Memoria Compartida
(shared memory)



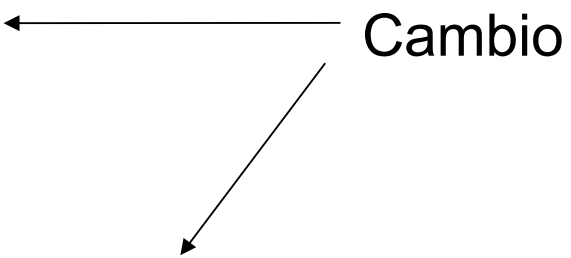
Suma en Paralelo
Combinando Hilos y Bloques

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

#define N 128*256
#define THREADS_PER_BLOCK 256
#define N_BLOCKS N/THREADS_PER_BLOCK

// Kernel to add N integers using threads and blocks
__global__ void add(int *a, int *b, int *c){
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  c[index] = a[index] + b[index];
}

// Main program
int main(void){
  ...
  ...

  add <<< N_BLOCKS , THREADS_PER_BLOCK >>> (a_dev,b_dev,c_dev);

  ...
  ...
}
```



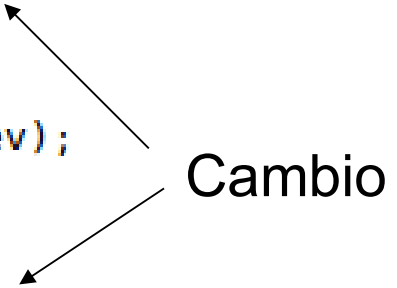
Producto Punto $c = \vec{a} \cdot \vec{b}$

```
// Main program
int main(void){
  int *a,*b,*c;               // Host copies
  int *a_dev,*b_dev,*c_dev;   // Device copies
  int size = N*sizeof(int);   // Size of N integer

  // Allocate host memory
  a = (int *) malloc (size);
  b = (int *) malloc (size);
  c = (int *) malloc (sizeof(int));

  // Allocate device memory
  cudaMalloc( (void**)&a_dev, size);
  cudaMalloc( (void**)&b_dev, size);
  cudaMalloc( (void**)&c_dev, sizeof(int));

  // Initialize
  for (int i=0; i<N; i++){
    a[i] = 1;
    b[i] = 1;
  }
  *c = 0;
```



Producto Punto

```
    // Copy inputs to device
    cudaMemcpy( a_dev, a, size        , cudaMemcpyHostToDevice );
    cudaMemcpy( b_dev, b, size        , cudaMemcpyHostToDevice );
    cudaMemcpy( c_dev, c, sizeof(int), cudaMemcpyHostToDevice );

    // Launch kernel on device
    dot <<< N_BLOCKS , THREADS_PER_BLOCK >>> (a_dev, b_dev, c_dev);

    // Copy device result back to host
    cudaMemcpy( c, c_dev, sizeof(int), cudaMemcpyDeviceToHost );

    // Free device memory
    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```



Kernel para el Producto Punto

```
// Kernel for dot product
__global__ void dot( int *a, int *b, int *c ) {

  __shared__ int prod[THREADS_PER_BLOCK]; // Shared memory
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  prod[threadIdx.x] = a[index] * b[index];

  __syncthreads();  // Threads synchronization

  if( threadIdx.x == 0) {
    int par_sum = 0;

    for(int i=0; i<THREADS_PER_BLOCK; i++)
      par_sum += prod[threadIdx.x]; // Threads reduction

    atomicAdd(c,par_sum); // Blocks reduction
  }
}
```

Este kernel contiene 3 elementos nuevos

Kernel para el Producto Punto

__shared__ int prod[THREADS_PER_BLOCK];

- Se hace uso de la memoria compartida, cada multiprocesador contiene una de estas memorias. Solo los hilos pueden compartir esta memoria.
- La memoria compartida es 2 ordenes de magnitud más rápida que la memoria global de la tarjeta aunque mucho más pequeña (entre 16KB y 48KB), es “como una memoria cache que uno puede controlar”.

__syncthreads();

- Cuando se quieren sumar los resultados de cada hilo se deben sincronizar todos los hilos, con el fin de estar seguros que ya estan almacenados en memoria todos los datos.
- Esto es lo que se conoce como una **operación bloqueante**: hasta que todos los hilos no lleguen al mismo punto no se puede avanzar en la ejecución del programa.

Operaciones Atómicas

atomicAdd(c,par_sum);

- Debido a que los bloques no se pueden comunicar entre sí (como lo hacen los hilos), entonces se utilizan las operaciones “atómicas” para hacer operaciones de reducción entre todos los bloques.
- Otra manera de hacerlo es almacenar el resultado de cada bloque en la memoria global y luego hacer la operación final en la CPU.

Otras operaciones atómicas disponibles:

- atomicAdd()
- atomicSub()
- atomicMin()
- atomicMax()
- atomicInc()
- atomicDec()
- atomicExch()
- atomicCAS()

Referencias

- Esta presentación y los programas que se usaron como ejemplo se encuentran en la página: http://www.ifm.umich.mx/~edison/cuda/
- Página oficial de Nvidia para todo lo que respecta a CUDA:

  http://developer.nvidia.com/category/zone/cuda-zone
- Conferencia de tecnología en GPU: http://www.gputechconf.com/page/home.html
- Libro de introducción a CUDA:

CUDA BY EXAMPLE

An introduction to general-purpose GPU programming.

Jason Sanders, Edward Kandrot

Gracias por su atención!!