MANEJO DE ERRORES Y ATOMICIDAD EN CUDA

Netz Romero

Manejo de errores

- Cuando se llaman las funciones de cuda, si éstas presentan algún problema regresan un tipo de error llamado `cudaError_t`
- En caso de que no suceda ningún tipo de error lo que regresan es: `cudaSuccess`

Veamos un ejemplo, en donde una función de cuda no cumple su prometido.

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

__global__ void add( int a, int b, int *c ) {
  *c = a + b;
}
int main( void ) {
  int c;
  int *dev_c;
  cudaMalloc((void**)&dev_c, 100000000000000 * sizeof(int));
  add<<<1,1>>>( 2, 7, dev_c );
  cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
  printf("2 + 7 = %d\n", c);
  cudaFree(dev_c);
  return 0;
}
```

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

__global__ void add( int a, int b, int *c ) {
  *c = a + b;
}
int main( void ) {
  int c;
  int *dev_c;
  cudaError_t error = cudaMalloc((void**)&dev_c,
                                 100000000000000 * sizeof(int));
  if(error != cudaSuccess) {
    // print the CUDA error message and exit
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }
  add<<<1,1>>>( 2, 7, dev_c );
  cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost);
  printf("2 + 7 = %d\n", c);
  cudaFree(dev_c);
  return 0;
}
```

```
#include <stdio.h>

__global__ void add( int a, int b, int *c ) {
  *c = a + b;
}

int main( void ) {
  int c;
  int *dev_c;
  cudaMalloc((void**)&dev_c, 100000000000000 * sizeof(int));
  add<<<1,1>>>( 2, 7, dev_c );
  cudaThreadSynchronize();
  cudaError_t error = cudaGetLastError();
  if(error != cudaSuccess) {
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }
  cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
  printf( "2 + 7 = %d\n", c );
  cudaFree( dev_c );
  return 0;
}
```

Manejando biblioteca de errores

- Se nombra el archivo error.h

```
static void HandleError( cudaError_t err,
                         const char *file,
                         int line ) {
  if (err != cudaSuccess) {
     printf("%s in %s at line %d\n“,cudaGetErrorString(err
),file,line);
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err )
(HandleError( err, __FILE__, __LINE__ ))
```

```
#include <stdio.h>
#include "error.h"

__global__ void add( int a, int b, int *c ) {
  *c = a + b;
}

int main( void ) {
  int c;
  int *dev_c;
  HANDLE_ERROR(cudaMalloc((void**)&dev_c,
                                  100000000000000*sizeof(int)));
  add<<<1,1>>>( 2, 7, dev_c );
  cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
  printf( "2 + 7 = %d\n", c );
  cudaFree( dev_c );
  return 0;
}
```

Comprobar valores del GPU

- Para comprobar los resultados los algoritmos elaborados para computo paralelo con el GPU, conviene analizar los resultados con un algoritmo elaborado secuencialmente en el CPU.
- Por ejemplo se corrobora los valores del producto punto calculados desde el GPU con los arrojados por el CPU.

```
#include<stdio.h>
#define N 4
__global__ void prodPunto(int a[], int b[], int *c) {
  __shared__ int prod[N];
  int i = threadIdx.x;
  if (i < N)
    prod[i] = a[i] * b[i];
  __syncthreads();
  if (threadIdx.x == 0) {
    int sum = 0;
    for (int k = 0; k < N; k++)
      sum += prod[k];
    *c = sum;
  }
}

__host__ void prodPuntoCPU(int a[], int b[], int *c) {
    int sum = 0;
    for (int i = 0; i < N; i++)
      sum += a[i] * b[i];
    *c = sum;
}
```

```
int main(void) {
 ...
 prodPunto<<<1, N>>>(a_d, b_d, c_d);
 ...
 prodPuntoCPU(a_h, b_h, &c_h_2);
 printf("El calculo por GPU es: %d\n", c_h_1);
 printf("El calculo por CPU es: %d\n", c_h_2);
 if(c_h_1 != c_h_2)
    printf("Diferencia de resultado entre GPU y CPU\n");
 else
    printf("No hay diferencia entre GPU y CPU\n");
 ...
  return 0;
}
```

Actividad

- Calcular el histograma de números aleatorios de 0 a 1024, bajo el paradigma de programación secuencial.

Compilando el programa de recursividad

```
$ nvcc deviceRecursivo.cu -o deviceRecursivo4
./deviceRecursivo.cu(8): Error: Recursive function
call is not supported yet: recursiva(int)
```

```
__global__ void histo_kernel( int *buffer,
                              int size,
                              int *histo ) {
int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        histo[buffer[i]]++;
         i += stride;
    }
}
```

```
int main( void ) {
 int *buffer = (int*)random_int( SIZE );
 int *dev_histo;
 ...
 cudaMemset( dev_histo, 0, SIZE * sizeof(int));
 histo_kernel<<<1, SIZE>>>( dev_buffer, SIZE, dev_histo);
 for (int i=0; i<SIZE; i++)
      histoCount += histo[i];
 printf( "Histograma GPU:  %ld\n", histoCount );
```

Atomicidad

- atomicAdd(&histo[buffer[i]], 1);
- nvcc hist_gpu.cu -arch sm_13 -o hist_gpu

Referencias

- Sito de NVIDIA, https://developer.nvidia.com/
- CUDA by Examples, NVIDIA
- Memoria compartida, http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/