Taller 4: CUDA

Malcolm Davis, Miembro Estudiantil, IEEE
mdavis.cr@ieee.org

Resumen— En este documento se encuentran las respuestas a las preguntas propuestas en el taller 4 del curso Arquitectura de Computadores II relacionadas con el paralelismo a nivel de datos de datos con CUDA.

Index Terms— TEC, Arquitectura De Computadores II, Threads, OpenMP, TLP, NEON, Arduino.

1. Micro-Investigación

1.1. ¿Qué es CUDA?

Cuda, según lo describen en la página de NVIDIA es una plataforma que permite el uso de los procesadores gráficos para el procesamiento general extendiendo los lenguajes existentes con palabras claves que se mapean en tiempo de compilación al GPU. [1]

1.2. ¿Qué es un kernel en CUDA y cómo se define?

Los kernels en CUDA son piezas de código que se van a ejecutar en los procesadores paralelamente. Se definen utilizando la palabra clave __global__ [2].

1.3. ¿Cómo se maneja el trabajo a procesar en CUDA? ¿Cómo se asignan los hilos y bloques?

El trabajo se maneja en bloques de hilos, y estos se asignan cuando se corre el kernel.

1.4. Investigue sobre la plataforma Jetson TX2 ¿Cómo está compuesta la arquitectura de la plataforma a nivel de hardware?

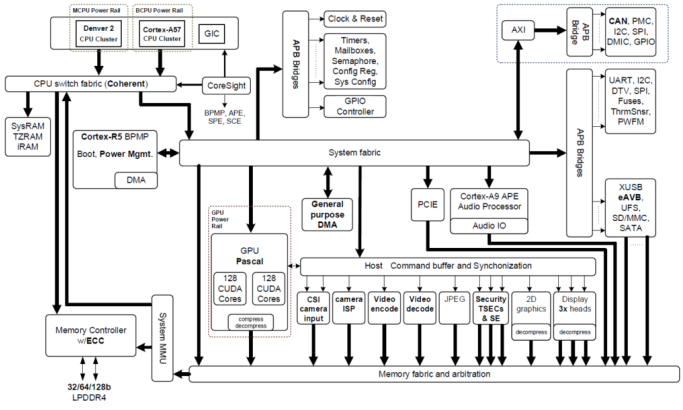

Figura 1. Arquitectura Jetson TX2

2. Ejemplo 1: Hola mundo en CUDA

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

int *a, *b;  // host data
int *c, *c2;  // results

//Cuda error checking - non mandatory
void cudaCheckError() {
 cudaError_t e=cudaGetLastError();
 if(e!=cudaSuccess) {
   printf("Cuda failure %s:%d: '%s'\n",
   __FILE__,__LINE__,cudaGetErrorString(e));
   exit(0);
 }
}

//GPU kernel
__global__
void vecAdd(int *A,int *B,int *C,int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

//CPU function
void vecAdd_h(int *A1,int *B1, int *C1, int N){
    for(int i = 0; i < N; i++)
        C1[i] = A1[i] + B1[i];
}

//Usage print
void usage(){
    printf("Usage:\n\
./vectadd for default parameters or;\n\
./vectadd threadsPerBlock N \n");
}

int main(int argc,char **argv)
{
    printf("Begin \n");
    //Iterations
    int n=10000000;
    //Number of blocks
    int nBytes = n*sizeof(int);
```

```
//Threads per block
int nTPB = 250;
//Block size and number
int block_size, block_no;

//memory allocation
a = (int *) malloc(nBytes);
b = (int *) malloc(nBytes);
c = (int *) malloc(nBytes);
c2 = (int *) malloc(nBytes);
//Getting args
if (argc > 1)
nTPB = atoi(argv[1]);
if (argc > 2)
n = atoi(argv[2]);
int *a_d,*b_d,*c_d;
block_size = nTPB; //threads per block
block_no = n/block_size;

printf("Using %d threads per block and n\
of size %d.\n", nTPB, n);

//Work definition
dim3 dimBlock(block_size, 1, 1);
dim3 dimGrid(block_no, 1, 1);

// Data filling
for(int i=0;i<n;i++)
a[i]=i,b[i]=i;

printf("Allocating device memory \
on host..\n");
//GPU memory allocation
cudaMalloc((void **) &a_d, n*sizeof(int));
cudaMalloc((void **) &b_d, n*sizeof(int));
cudaMalloc((void **) &c_d, n*sizeof(int));

printf("Copying to device..\n");
cudaMemcpy(a_d, a, n*sizeof(int),
cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, n*sizeof(int),
cudaMemcpyHostToDevice);

clock_t start_d=clock();
printf("Doing GPU Vector add\n");
vecAdd<<<block_no,block_size>>>(a_d,
b_d, c_d, n);
cudaCheckError();

//Wait for kernel call to finish
cudaThreadSynchronize();

clock_t end_d = clock();


printf("Doing CPU Vector add\n");
clock_t start_h = clock();
vecAdd_h(a, b, c2, n);
    clock_t end_h = clock();

//Time computing
    double time_d = (double)(end_d-start_d)/
    CLOCKS_PER_SEC;
    double time_h = (double)(end_h-start_h)/
    CLOCKS_PER_SEC;

    //Copying data back to host, this is
    //a blocking call and will not start
    //until all kernels are finished
    cudaMemcpy(c, c_d, n*sizeof(int),
    cudaMemcpyDeviceToHost);
    printf("n = %d \t GPU time = %fs \
    \t CPU time = %fs\n", n, time_d, time_h);

    //Free GPU memory
    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(c_d);
    return 0;
}
```

2.1. Analice el código vecadd.cu y el Makefile correspondiente. A partir del análisis del código, extraiga cuáles son los pasos generales para la generación de aplicaciones utilizando CUDA

Los pasos principales son:

- Declarar el Kernel
- Declarar y reservar memoria
- Inicializar los datos en el CPU
- Transferir los datos del CPU al GPU
- Ejecutar el Kernel declarado anteriormente
- Transferir los resultados del GPU al CPU

2.2. Analice el código fuente del kernel vecadd.cu. A partir del análisis del código, determine ¿Qué operación se realiza con los vectores de entrada? ¿Cómo se identifica cada elemento a ser procesado en paralelo y de qué forma se realiza el procesamiento paralelo?

Se puede observar en el kernel que la operación que se realiza sobre los vectores es una suma de vectores:

```
//GPU kernel
__global__
void vecAdd(int *A,int *B,int *C,int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

En la cuarta linea de este código se aprecia la operación. Para determinar los elementos se utiliza la linea 3 del código y para empezar el procesamiento en paralelo se utiliza la linea:

```
vecAdd<<<block_no,block_size>>>(a_d,
b_d, c_d, n);
```

2.3. Realice la ejecución de la aplicación vecadd. ¿Qué hace finalmente la aplicación?

Como se mencionó anteriormente, la aplicación hace una suma de vectores.

2.4. Cambie la cantidad de hilos por bloque y el tamaño del vector. Compare el desempeño antes al menos 5 casos diferentes.

REFERENCIAS

[1] M. Ebersole, *"What Is CUDA — NVIDIA Official Blog"*. The Official NVIDIA Blog, 2012. [Online]. Available: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/. [Accessed: 13- May- 2018].

[2] M. Harris, *.An Easy Introduction to CUDA C and C++"*. NVIDIA Developer Blog, 2012. [Online]. Available: https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/. [Accessed: 13- May- 2018].