Área Académica en Computadores

Curso

CE4302 - Arquitectura de Computadores II

Taller 4

Profesor: Ing. Luis Barboza Artavia

Grupo: 1

Tema:

CUDA

Elaborado por

Juan Ignacio Navarro - 2019039662

Jose David Sánchez - 2018142388

Mónica Waterhouse - 2017076143

Jose Ignacio Granados - 2018319698

Octubre 2022 Cartago, Costa Rica

Investigación:

1. ¿Qué es CUDA?

   Esta corresponde a una plataforma de computación paralela y también un modelo de programación creado por NVIDIA. CUDA ayuda a mejorar el uso de los aceleradores GPU. Se suele confundir con un lenguaje de programación o un API, pero es más que eso pues con CUDA se pueden desarrollar bibliotecas, SDKs y herramientas para optimización [1].

   Cuda se puede ver como una abstracción de las arquitecturas GPU que actúa como un puente entre una aplicación y su posible implementación en el hardware de GPU [1].

2. ¿Qué es el kernel en CUDA y cómo se define?

   El kernel en CUDA es una función que se ejecuta en la GPU. La parte paralela de las aplicaciones se ejecuta K veces en paralelo por K hilos de CUDA diferentes [2].

   Un grupo de hilos se llama bloque de CUDA. Los bloques de CUDA se agrupan en una malla. Un kernel es ejecutado como una malla de bloques de hilos.

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

   Cada bloque CUDA se ejecuta por un Multiprocesador streaming (SM) y no puede ser migrado por otros SMs en la GPU. Un SM puede correr varios bloques de CUDA concurrentes dependiendo de los recursos necesarios por el bloque de CUDA. Cada kernel es ejecutado en un dispositivo y CUDA soporta correr varios kernels en un dispositivo al mismo tiempo. Todo esto se observa en la figura 1 [2].

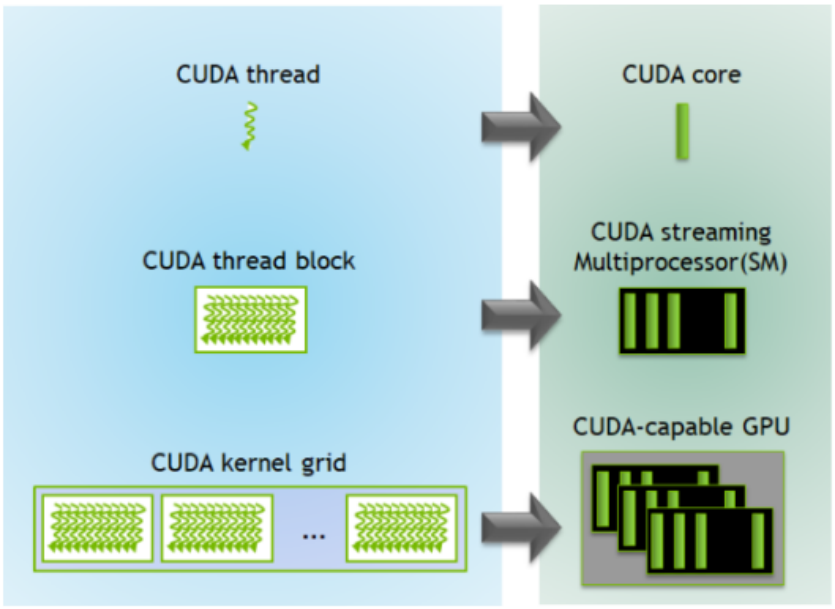

Figura 1. Ejecución en CUDA.

4. Investigue sobre la plataforma Jetson Nano, ¿cómo está compuesta la arquitectura de la plataforma nivel de hardware?

   A nivel de hardware se tienen las especificaciones mostradas en la figura 2 [3].

Developer Kit Technical Specifications	
GPU	128-core NVIDIA Maxwell™
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4GB 64-bit LPDDR4 25.6GB/s
Storage	microSD (Card not included)
Video Encoder	4Kp30 | 4x 1080p30 | 9x 720p30 (H.264/H.265)
Video Decoder	4Kp60 | 2x 4Kp30 | 8x 1080p30 | 18x 720p30 (H.264/H.265)
Connectivity	Gigabit Ethernet, M.2 Key E
Camera	2x MIPI CSI-2 connectors
Display	HDMI and DP
USB	4x USB 3.0, 1x USB 2.0 Micro-B
Others	40-pin header (GPIO, I2C, I2S, SPI, UART) 12-pin header (Power and related signals, UART) 4-pin fan header
Mechanical	100 mm x 80 mm x 29 mm

Figura 2. Especificaciones del hardware de la Jetson Nano.

5. Cómo se compila un código CUDA?

   El compilador de CUDA C, nvcc, que es parte del toolkit de NVIDIA CUDA compila un código con extensión .cu como por ejemplo **saxpy.cu** de la forma

   - nvcc -o saxpy saxpy.cu

   Luego para correr el código se aplica:

   - ./saxpy

Análisis:

1. Analice el código vecadd.cu

En el main de este código se tienen las definiciones de las variables a utilizar tanto en las prueba con el GPU como para con el CPU:

```
29  int main(int argc,char **argv)
30  {
31      printf("Begin \n");
32      //Iterations
33      int n=10000000;
34      //Number of blocks
35      int nBytes = n*sizeof(int);
36      //Block size and number
37      int block_size, block_no;
38
39      //memory allocation
40      a = (int *) malloc(nBytes);
41      b = (int *) malloc(nBytes);
42      c = (int *) malloc(nBytes);
43      c2 = (int *) malloc(nBytes);
```

Donde la idea es que se realice una suma del vector a con el b y su resultado se almacene en c para la ejecución en GPU y en c2 para la ejecución en el CPU.

Luego se definen las variables que se usan exclusivamente para la ejecución en el GPU

```
45      int *a_d,*b_d,*c_d;
46      block_size = 250; //threads per block
47      block_no = n/block_size;
```

En este caso se definió que se utilizarán 250 hilos por bloque, lo cual cambia la cantidad de bloques. Estos datos de los bloques se necesitan para llamar a las funciones que se ejecutarán en la GPU pues depende de conocer cómo se debe aplicar el paralelismo.

Luego se define la dimensión del bloque y el grid.

```
49      //Work definition
50      dim3 dimBlock(block_size, 1, 1);
51      dim3 dimGrid(block_no, 1, 1);
```

Después se llena el vector a y el b con números consecutivos en cada uno de sus elementos, empezando desde cero:

```
53      // Data filling
54      for(int i=0;i<n;i++)
55      a[i]=i,b[i]=i;
56
```

Ahora se reserva el espacio en el device y se copia desde el host al device para poder hacer las operaciones:

```
58      printf("Allocating device memory on host..\n");
59    //GPU memory allocation
60      cudaMalloc((void **) &a_d, n*sizeof(int));
61      cudaMalloc((void **) &b_d, n*sizeof(int));
62      cudaMalloc((void **) &c_d, n*sizeof(int));
63
64      printf("Copying to device..\n");
65      cudaMemcpy(a_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
66      cudaMemcpy(b_d, b, n*sizeof(int), cudaMemcpyHostToDevice);
67
```

Por último se realizan ambas operaciones en la GPU y luego en el CPU, para ambas se toman los tiempos de ejecución para posterior análisis. Note que las funciones realizan lo mismo, solamente que una lo hace de forma secuencial en un for y la otra en paralelo.

```
68      clock_t start_d=clock();
69      printf("Doing GPU Vector add\n");
70      vecAdd<<<block_no,block_size>>>(a_d, b_d, c_d, n);
71      cudaCheckError();
72
73      //Wait for kernel call to finish
74      cudaThreadSynchronize();
75
76      clock_t end_d = clock();
77      
78
79      printf("Doing CPU Vector add\n");
80      clock_t start_h = clock();
81      vecAdd_h(a, b, c2, n);
82      clock_t end_h = clock();
```

```
16  //GPU kernel
17  __global__
18  void vecAdd(int *A,int *B,int *C,int N){
19      int i = blockIdx.x * blockDim.x + threadIdx.x;
20      C[i] = A[i] + B[i];
21  }
22
23  //CPU function
24  void vecAdd_h(int *A1,int *B1, int *C1, int N){
25      for(int i = 0; i < N; i++)
26          C1[i] = A1[i] + B1[i];
27  }
```

2. Analice el código fuente del kernel vecadd.cu. A partir del análisis del código, determine:
    a. ¿Qué operación se realiza con los vectores de entrada?

Este código corresponde al de la última imagen, en las líneas 17 a 20. Note que en la 19 lo que se realiza es obtener el índice a afectar para cada hilo [4]. Las variables:

blockIdx.x: índice del bloque dentro del grid.

blockDim.x: contiene las dimensiones de cada bloque de hilo.

threadIdx.x: es el índice del hilo dentro del bloque.

3. Realice la compilación del código vecadd.cu

```
arqui2@arqui2-desktop:~/Downloads/Taller4_CUDA/src$ nvcc -o vecadd vecadd.cu
vecadd.cu: In function 'int main(int, char**)':
vecadd.cu:74:23: warning: 'cudaError_t cudaThreadSynchronize()' is deprecated [-
Wdeprecated-declarations]
     cudaThreadSynchronize();
                       ^
/usr/local/cuda/bin/../targets/aarch64-linux/include/cuda_runtime_api.h:957:46:
note: declared here
 extern __CUDA_DEPRECATED __host__ cudaError_t CUDARTAPI cudaThreadSynchronize(v
oid);
                                               ^~~~~~~~~~~~~~~~~~~~
```

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

```
arqui2@arqui2-desktop:~/Downloads/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000     GPU time = 0.000410     CPU time = 0.196332
arqui2@arqui2-desktop:~/Downloads/Taller4_CUDA/src$ gedit vecadd.cu
```

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

Hilos por bloque	**Tamaño del vector**	**Tiempo en CPU**	**Tiempo en GPU**
250	10000000	0.196332	0.000410
25	1000	0.000014	0.000125
500	5000000	0.180810	0.000618
300	7500000	0.101254	0.000208
100	10000000	0.267177	0.000394

Se puede observar claramente que el tamaño del vector y la cantidad de hilos por bloque van a representar un papel importante en el tiempo total de ejecución. Para tener un punto de comparación más amplio se decide en la primer y última iteración dejar un mismo tamaño de vector, en este caso de 10000000, y luego solamente modificar el número de hilos por bloque resultando en una mejor poco significativa ya que la curva de paralelización tiende a ser asintótica hasta por lo que si se siguen aumentando los hilos no se verá mayor mejora.

Se puede concluir que en la mayoría de los casos implementar hilos va a ser una buena opción, todo va a depender de la función que se quiera paralelizar y la cantidad de iteraciones que planeen hacer. Un caso en el que esto no se cumple es en la segunda ejecución en la cual se utiliza un vector muy pequeño y pocos hilos. Al poseer estas características, implica para el dispositivo un mayor trabajo implementar hilos y luego unir sus resultados.

Ejercicios prácticos

1. Realice un programa que calcule el resultado de la multiplicación de dos matrices de 4x4, utilizando el paralelismo con CUDA.

Para la implementación de este programa se va a realizar la multiplicación de dos matrices de tamaño 4x4 en el archivo llamado “mult_mat.cu”. La idea es que se implemente la paralelización de operaciones por medio de la arquitectura de CUDA la cual permite poseer gran cantidad de hilos para el cálculo de los valores para la matriz resultante. Para esto primero se debe reservar la memoria de las matrices por medio de “cudaMallocManaged” como se muestra en la imagen a continuación.

```
// Allocate Unified Memory – accessible from CPU or GPU
cudaMallocManaged(&A, A_size*sizeof(float));
cudaMallocManaged(&B, B_size*sizeof(float));
cudaMallocManaged(&C, C_size*sizeof(float));
cudaMallocManaged(&C_cpu, C_size*sizeof(float));
```

Seguidamente, se inicializan las matrices con valores random por medio de una función llamada “initialize_matrix” rellenando las matrices con los valores previamente creados y que son pasados como argumentos de la función para su uso como se muestra a continuación.

```
// initialize A and B matrices
auto all_ones = []() -> float {
  return 1.0f;
};

srand (time(NULL));
auto rand_numbers = []() -> float {
  auto f = static_cast<float>(rand())/(static_cast<float>(RAND_MAX/1000));
  int n = static_cast<int>(f);
  return static_cast<float>(n);
};

initialize_matrix<float>(A, A_rows, A_cols, rand_numbers);
initialize_matrix<float>(B, B_rows, B_cols, rand_numbers);
```

```
template<typename T>
void initialize_matrix(T* M, int rows, int cols, std::function<float()> F) {
  for(int i = 0; i < rows; i++){
    for(int j = 0; j < cols; j++){
      M[i * cols + j] = F();
    }
  }
}
```

El siguiente paso se encarga de definir el tamaño del “grid” que va a contener los bloques con los hilos. Además también se debe definir el tamaño del bloque. Esto para luego ejecutar la función llamada “naive_matrix_multiply” que multiplica las matrices y guarda los resultados en una tercera matriz previamente definida. Este proceso se muestra a continuación.

```
dim3 dim_grid(C_cols/COL_TILE_WIDTH, C_rows/ROW_TILE_WIDTH, 1);
dim3 dim_block(COL_TILE_WIDTH, ROW_TILE_WIDTH, 1);


naive_matrix_multiply<float><<<dim_grid, dim_block>>>(A, B, C, A_cols, C_rows, C_cols);
```

```
template<typename T>
__global__
void naive_matrix_multiply(T *A, T *B, T* C, int width, int C_rows, int C_cols)
{
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  // check boundry conditions
  if( row < C_rows && col < C_cols ){
    // do the multiplication for one row and col
    T value = 0;
    for(int k = 0; k < width; k++){
      value += A[row * width + k] * B[k * C_cols + col];
    }
    // store result
    C[row * C_cols + col] = value;

  }
```

Para finalizar se sincronizan los resultados con la función “cudaDeviceSynchronize()” para proceder a imprimir todos los valores en la terminal y luego liberar la memoria reservada como se muestra a continuación.

```
// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();

/* Printing the contents of third matrix. */
printf("\n\nA matrix :\n");
for(int i=0; i< 4*4; i++){
  //printf("\n\t\t\t");
  //for(int j=0; j < 4; j++)
  printf("%f\t", A[i]);}

/* Printing the contents of third matrix. */
printf("\n\nB matrix :\n");
for(int i=0; i< 4*4; i++){
  //printf("\n\t\t\t");
  //for(int j=0; j < 4; j++)
  printf("%f\t", B[i]);}

/* Printing the contents of third matrix. */
printf("\n\nResultant matrix :\n");
for(int i=0; i< 4*4; i++){
  //printf("\n\t\t\t");
  //for(int j=0; j < 4; j++)
  printf("%f\t", C[i]);}
printf("\n\nFinished\n");
```

2. Proponga una aplicación que involucre procesamiento vectorial. Implemente dicha aplicación tanto serial (sin paralelismo) como con CUDA. Mida tiempos de ejecución para diferentes tamaños y/o iteraciones.

Para la aplicación que involucra procesamiento vectorial se va a implementar la función “saxpy” en el archivo llamado “saxpy.cu”. Dicha función se basa en una multiplicación escalar y suma de vectores. Es por esto que es altamente paralelizable si se implementan las herramientas de CUDA gracias a sus múltiples hilos. La idea es que se implemente dicha función con y sin paralelización y luego comparar los resultados. El primer paso es reservar las memorias tanto para el CPU como para el GPU. Esto se observa en la imagen a continuación.

```
int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Seguidamente se procede a ejecutar la función “saxpy” para un millón de elementos con su característica paralelizable gracias a “CUDA” tomando su tiempo de ejecución. Esto se muestra en las imágenes a continuación.

```
// Perform SAXPY on 1M elements
clock_t start_d=clock();
printf("Doing GPU Vector SAXPY\n");
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
clock_t end_d = clock();
```

```
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Luego se procede a ejecutar la función “saxpy_h” la cual no posee la característica paralelizable por lo cual se espera que sea más lenta a la hora de comparar los resultados. Este paso se muestra en la imágenes a continuación.

```
printf("Doing CPU Vector add\n");
clock_t start_h = clock();
saxpy_h(N, 2.0f, x, y);
clock_t end_h = clock();
```

```
//CPU function
void saxpy_h(int n, float a, float *x, float *y){
    for(int i = 0; i < n; i++)
        y[i] = a*x[i] + y[i];
}
```

Para finalizar, se imprimen los resultados para ser comparados y se libera la memoria. Esto se muestra en la imagen a continuación.

```
//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
printf("\t GPU time = %f \t CPU time = %f\n", time_d, time_h);

cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

Referencias:

[1] Fred, O. (2012) *What is CUDA?*. Disponible en: What Is CUDA | NVIDIA Official Blog

[2] Gupta, P. (2020) *CUDA Refresher: The CUDA Programming Model*. Disponible en: CUDA Refresher: The CUDA Programming Model | NVIDIA Technical Blog

[3] NVIDIA (s.f.) *Jetson Nano*. Disponible en: Jetson Nano | NVIDIA Developer

[4] Harris, M. (2012) *An easy Introduction to CUDA C and C++*. Disponible en: An Easy Introduction to CUDA C and C++ | NVIDIA Technical Blog