Pensando en paralelo

Francisco Hernández López,
Joel Trejo Sánchez y
Miguel Uh Zapata

Centro de Investigación en Matemáticas, A. C.,
Unidad Mérida

Verano de Investigación
Julio 2020

Resumen

Muchos problemas complejos del cómputo científico y matemático han sido resueltos en las últimas décadas gracias a las herramientas computacionales con las que contamos en la actualidad. Entre estas herramientas está el cómputo en paralelo. En este curso se explorará el pensamiento que hay detrás de programar un algoritmo en paralelo y en qué se diferencia del desarrollo de códigos estándar en serie. Como muestra se diseñará un algoritmo para la resolución de un simple problema de cálculo aritmético, pero que ejemplifica claramente el potencial y la necesidad del uso de la paralelización. Dicho algoritmo se implementará y estudiará en paralelo utilizando tres enfoques diferentes: memoria compartida, paso de mensajes y tarjetas gráficas.

Índice

Pensando en paralelo

En este curso se proponen tres paradigmas de programación paralela. El paradigma utilizando memoria compartida, por medio de OpenMP; el paradigma utilizando el paso de mensajes, por medio de MPI; y el paradigma utilizando tarjetas de video, por medio de CUDA. A continuación se presentará algunos detalles y ejemplos de cada uno de estos temas que ayudarán a entender mejor la idea detrás de cada uno de estos temas.

1. Algoritmos en paralelo en memoria compartida: OpenMP

OpenMP es una interfaz que se utiliza directamente para programación multi-hilo y utilizando memoria compartida. Permite una interfaz simple para el desarrollo de aplicaciones paralelas mediante directivas al compilador [1].

Una de las principales ventajas de utilizar OpenMP, es que hoy en día casi todas las computadoras tienen procesadores multi-nucleo (multi-core). Por lo tanto, basta con configurar el compilador para utilizar OpenMP. En esta sección utilizaremos OpenMP.

1.1. Instalación

La dificultad de instalar y configurar OpenMP para su uso depende del sistema operativo a utilizar. Para el caso de *Linux*, el sistema operativo ya está preparado el compilador para utilizar OpenMP. Para el caso de Windows o Mac se deben realizar diversas acciones para habilitar la posibilidad de utilizar OpenMP como se explica a continuación:

- En Windows, sugerimos instalar CodeBlocks y utilizar el compilador MinGW. Se propone realizar los pasos descritos en la el siguiente link para lograr la configuración:
  "http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/tutoriales/openmp/index.html"
- Utilizando el sistema operativo MAC, la versión incluida en el XCode, no es compatible con OpenMP, por lo que hay que realizar algunos ajustes, posteriores a la instalación de XCode (en caso que no lo tenga instalado). Para configurar OpenMP en MacOS Sierra se sugiere revisar el siguiente link:
  "https://iscinumpy.gitlab.io/post/omp-on-high-sierra/"
  Es posible encontrar tutoriales similares al anterior dependiendo de la versión de MacOS con la que cuente el usuario.

1.2. Ejemplos OpenMP

Ahora, describimos tres problemas muy sencillos, que nos permitirán comprender la utilidad de OpenMP. Los códigos pueden descargarse del siguiente link:

`https://github.com/trejoel/PensandoParalelo2020`

1.2.1. Mi primer programa en paralelo usando OpenMP

Este primer programa es el clásico hola mundo ó hello world por sus siglas en inglés. Este programa consiste en pedirle al compilador que imprima un mensaje de texto (en este caso un "hola mundo") en la salida estándar. En el caso de OpenMP, debido a que contamos con varios nucleos (cores) en los procesadores, podemos instruir al procesador que cada uno de dichos cores imprima un mensaje en pantalla.

El siguiente código imprime un hola mundo por cada nucleo del procesador.

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

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

int numero_hilos, id_hilo;
#pragma omp parallel private(numero_hilos, id_hilo)
{
numero_hilos=omp_get_num_threads();
id_hilo=omp_get_thread_num();
if (id_hilo==0){
printf("Este computador utiliza %d cores\n",numero_hilos);
}
printf("Soy el core n\'umero %d\n",id_hilo);
}
}
```

Nótese la llamada a la librería "omp.h" en la sección #include del programa. Esta instrucción le indica al compilador que el programa realizará llamadas a funciones propias de OpenMP.

Dicho algoritmo imprime un mensaje en pantalla por cada uno de los nucleos del procesador. En particular, para el primer nucleo además se imprime un mensaje adicional que indica el número de nucleos con el que cuenta el procesador.

Para compilar el programa se utiliza el compilador `-fopennmp` de la siguiente manera:

```
gcc -fopennmp hello.c -o hello
```

1.2.2. Suma de dos vectores de dimensión N

El siguiente problema tiene que ver con la suma de vectores. La siguiente función recibe como entrada un par de vectores A y B de dimensión N, y regresa como salida un vector C de dimensión N que representa la suma de dichos vectores. Es decir.

- Dados $\vec{A} = \{a_1, \ldots, a_N\}$ y $\vec{B} = \{b_1, \ldots, b_N\}$, el vector suma está dado por

$$\vec{C} = \vec{A} + \vec{B} = \{c_1, \ldots, c_N\},$$

  donde $c_i = a_i + b+_i$, para $a_i \in \vec{A}$ y $b_i \in \vec{B}$.

La siguiente función realiza la suma secuencial de los vectores.

```c
double *addArraySeq(double *A, double *B, int numberOfElements){
double *C;
C=malloc(numberOfElements*sizeof(double));
double wtime=0;
for (int i=0;i<numberOfElements;i++){
C[i]=A[i]+B[i];
}
return C;
}
```

En el ciclo `for` se realiza la suma de cada elemento de los vectores $\vec{A}$ y $\vec{B}$, para obtener el correspondiente vector $\vec{C}$.

Para paralelizar el código basta con agregar un par de líneas de código adicionales. La siguiente función muestra el código en paralelo:

```c
double *addArrayPar(double *A, double *B, int numberOfElements){
double *C;
C=malloc(numberOfElements*sizeof(double));
double wtime=0;
wtime = omp_get_wtime();
```

```
6  #pragma omp parallel for
7  for (int i=0;i<numberOfElements;i++){
8  C[i]=A[i]+B[i];
9  }
10 wtime = omp_get_wtime ()-wtime;
11 printf("Tiempo de respuesta paralelo:%f\n",wtime);
12 return C;
13 }
```

1.2.3. Suma de los elementos de un vector de dimensión N

El tercer ejemplo es el de mayor complejidad de programar en paralelo entre los tres presentados en este material. En este problema se recibe un vector con N elementos y realiza la suma de todos los elementos en el vector.

En el siguiente código, la función `addVectorElements` realiza la suma en forma secuencial, y la función `addVectorElementsPar` realiza la forma en paralelo utilizando el total de núcleos del procesador. Nótese que en este ejemplo utilizamos la cláusula `reduce`. Un ejercicio interesante sería no utilizar dicha cláusula y observar el comportamiento de la función `addVectorElementsPar`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <omp.h>
5
6  #define bool int
7  #define true 1
8  #define false 0
9
10 double *generateRandomArray(int numberOfElements);
11 double addVectorElements(double *A, int numberOfElements);
12 double addVectorElementsPar(double *A, int numberOfElements);
13
14 int main(int argc, char *argv[]){
15    double *A;
16    double sum;
17    int n;
18    printf("De cuantos elementos son los arreglos\n");
19    scanf("%d",&n);
20    A=generateRandomArray(n);
21    sum=addVectorElements(A,n);
22    printf("El valor de la suma secuencial es %f:\n",sum);
23    sum=addVectorElementsPar(A,n);
24    printf("El valor de la suma paralela es %f:\n ",sum);
25    free(A);
26 }
27
28 double *generateRandomArray(int numberOfElements){
29    double *myArray;
30    srand(time(NULL));
31    myArray=malloc(numberOfElements*sizeof(double));
32    for (int i=0;i<numberOfElements;i++){
33        myArray[i]=rand() %100 + 1;
34    }
35    return myArray;
36 }
37
38 double addVectorElements(double *A, int numberOfElements){
39    double wtime=0;
40    double sum=0;
41    wtime = omp_get_wtime ();
42    for (int i=0;i<numberOfElements;i++){
43         sum=sum+A[i];
44    }
45   wtime = omp_get_wtime ()-wtime;
46   printf("Tiempo de respuesta secuencial:%f\n",wtime);
47    return sum;
48 }
49
50
51 double addVectorElementsPar(double *A, int numberOfElements){
```

```
52      double wtime=0;
53      double sum=0;
54      wtime = omp_get_wtime ();
55      #pragma omp parallel for reduction (+:sum)
56      for (int i=0;i<numberOfElements;i++){
57           sum=sum+A[i];
58      }
59     wtime = omp_get_wtime ()-wtime;
60     printf("Tiempo de respuesta paralelo:%f\n",wtime);
61      return sum;
62  }
```

Finalmente, en esta sección de este curso explicaremos otras claúsulas más complejas que permitirán realizar programas en paralelo más sofisticados.

2. Algoritmos en paralelo en entornos multicore: MPI

En esta sección se describen los conceptos básicos para la programación en paralelo usando el entorno de MPI y se presentan algunos ejemplos numéricos para medir el rendimiento del código en paralelo.

MPI (*"Message Passing interface"*) está definida como una especificación para el estándar de la librería de paso de mensajes según el foro oficial de MPI (http://mpi-forum.org). El modelo de de computación paralela de paso de mensajes se caracteriza principalmente en que cada proceso tiene su propia memoria local, pero son capaces de comunicarse entre sí enviando y recibiendo mensajes.

2.1. Conceptos básicos de MPI

Aclaremos que MPI no es un lenguaje de programación, es una librería, especifíca los nombres y las llamadas de funciones o subrutinas para programas en Fortran, C o C++, dichos programas pueden ser compilados con los compiladores ordinarios de cada lenguaje y únicamente necesitan ser vinculados con la librería MPI [2].

Para empezar a desarrollar un programa con MPI es necesario entender los conceptos básicos del modelo de paso de mensajes. Cada proceso puede ejecutar un código de manera "normal", como si estuviese ejecutando un código en serie. Sin embargo, en un entorno paralelo de paso de mensajes, son varios los procesos que están ejecutando dicho código, no obstante, puede estén ejecutando el mismo algoritmo pero con datos diferentes, por lo que es importante que puedan comunicar sus resultados locales con los demás procesos. Para esto es necesario poder identificar a los procesos de alguna manera y saber cuántos de ellos son.

> OBSERVACIÓN:
> Es importante aclarar que el número de procesos no es lo mismo que procesadores, los procesadores son los elementos electrónicos que tiene el sistema físicamente, mientras que los procesos, es la cantidad de tareas que se ejecutarán en paralelo. Dicha cantidad lo define el usuario a la hora de correr el programa. Esto quiere decir que, aunque un sistema cuente con 8 procesadores físicos, se puede correr un programa con únicamente un proceso, que sería equivalente a correrlo de forma serial. También se puede correr usando dos procesos, de esta manera, cada proceso es asignado a un procesador, por lo que el programa se correría en paralelo usando dos procesadores físicos, así sucesivamente al ir aumenta el número de procesos. Pero ¿qué es lo que pasa cuando el número de procesos excede al número de procesadores?

Como ya se ha mencionado, para que el modelo de paso de mensajes tenga sentido, es necesario

poder identificar a los procesos, saber cuántos son y poder enviar y recibir mensajes entre ellos. Por ello es que existen estas seis funciones básicas:

```
MPI_INIT
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize
```

El primer y último comando son únicamente para inicializar y finalizar, respectivamente, el entorno de MPI. Con `MPI_Comm_size` se puede conocer el número de procesos que se están ejecutando. `MPI_Comm_rank` devuelve el identificador del proceso, es un número entero que va desde cero hasta el número de procesos menos uno. Con `MPI_Send` y `MPI_Recv` se le ordena a un proceso que envíe o reciba un mensaje respectivamente, estas son las funciones básicas para el intercambio de datos. Con únicamente estas funciones básicas, uno ya es capaz de desarrollar un programa en MPI. De hecho, la mayoría de las demás funciones o subrutinas que existen en MPI, son únicamente herramientas para facilitar la comunicación entre procesos, es decir, que se podría realizar casi todo lo que realiza cualquier otra función o subrutina utilizando únicamente estas seis funciones. Debido al corto del tiempo de este curso no podremos estudiar las funciones de comunicación.

2.2. Instalación

Para poder ejecutar programas en paralelo usando MPI es necesario instalar MPICH. En la siguiente liga se puede encontrar una guia que les puede ayudar en el proceso de instalación de este programa:

https://www.mpich.org/static/downloads/3.3.2/mpich-3.3.2-installguide.pdf

Tanto los procedimientos para Unix y Windows pueden ser encontradas en esta guía.

2.3. Ejemplos MPI

A continuación presentaremos una serie de ejemplos que nos ayudarán a entender mejor los conceptos de esta forma de programar en paralelo.

2.3.1. Mi primer programa en paralelo usando MPI

Al igual que el caso de OpenMP, nuestro primer programa es la versión MPI del clásico hola mundo ó hello world. Este programa consiste en pedirle al compilador que imprima un mensaje de texto en la salida estándar. En el caso de MPI, debido a que contamos con varios procesos ejecutados al mismo tiempo, cada uno imprimirá un mensaje en pantalla.

El siguiente código imprime un `Yo soy el procesor` por cada proceso.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  int main()
6  {
7     int numtasks, taskid;
8     //------------------------
9     MPI_Init(NULL,NULL);                    /* Initializacion*/
10    MPI_Comm_size(MPI_COMM_WORLD,&numtasks); /* Numero de procesos */
11    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);   /* Identificador de cada proceso */
12    //------------------------
13    printf(" Yo soy el procesor %i de %i \n",taskid,numtasks);
14    //------------------------
15    MPI_Finalize();
```

```
16      //-----------------------
17      return 0;
18  }
```

Nótese la llamada a la librería “mpi.h” en la sección #include del programa. Esta instrucción le indica al compilador que el programa realizará llamadas a funciones propias de MPI.

Para ejecutar el programa es necesario primero compilar. A continuación se indica de manera específica como ejecutar el programa con 4 procesos:

```
mpicxx HelloMPI.cpp -o HelloMPI
mpiexec -n 4 ./HelloMPI
```

Dicho algoritmo imprime un mensaje en pantalla por cada uno de los procesos indicando su número asignado y el total de procesos. Notar que el número empieza en cero:

```
1  Yo soy el procesor 0 de 4
2  Yo soy el procesor 2 de 4
3  Yo soy el procesor 3 de 4
4  Yo soy el procesor 1 de 4
```

2.3.2. Suma de dos vectores de dimensión N

El siguiente problema suma de vectores de longitud N. En este caso ambos vectores fueron generados de manera aleatoria usando el comando `rand`. A continuación se describe el programa en paralelo usando MPI.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4   #include <time.h>
5   #include <mpi.h>
6
7   double *generateRandomArray(int numberOfElements);
8
9   int main()
10  {
11      int N = 10;
12      int i;
13      double cpu_time_used;
14      double *A,*B,*C;
15      //=========================================
16      // MPI [1]: definiciones
17      int numtasks, taskid;
18      int Nx,Ndom,iIni,iFin;
19
20      //-----------------------------------------
21      // Vector de N elementos
22      A = generateRandomArray(N);
23      B = generateRandomArray(N);
24      C = (double *)malloc(N*sizeof(double));
25
26      //=========================================
27      // MPI [2]: inicio
28      MPI_Init(NULL,NULL);                        /* Initializacion */
29      MPI_Comm_size(MPI_COMM_WORLD,&numtasks); /* Numero de procesos */
30      MPI_Comm_rank(MPI_COMM_WORLD,&taskid);   /* Identificador de cada proceso */
31
32      //=========================================
33      // MPI [3]: Division del trabajo
34      Ndom = (int)(1.0*N/numtasks);
35      if (numtasks == 1) {
36          Nx = N;
37      }
38      else {
39          if(taskid==numtasks-1){
40              Nx = N - Ndom*(numtasks-1);
41          }
42          else {
```

```
43              Nx = Ndom;
44          }
45      }
46      iIni = taskid*Ndom;
47      iFin = iIni + Nx;
48
49      //-----------------------------------------
50      // Operaciones: sumar dos vectores
51      clock_t tiempo_ejec;
52      double start = MPI_Wtime();
53      //------------------------
54      for (i = iIni; i < iFin; i++) {
55           C[i] = A[i] + B[i];
56           printf("%f + %f = %f\n",A[i],B[i],C[i]);
57      }
58      //------------------------
59      double end = MPI_Wtime();
60      cpu_time_used = (double)(end-start);
61
62      //-----------------------------------------
63      // Imprimir
64      printf("Procesor %i de %i: \n \
65              Nx     = %i  \n \
66              iIni   = %i  \n \
67              iFin   = %i  \n \
68              time   = %f \n \n",taskid,numtasks,Nx,iIni,iFin,cpu_time_used);
69
70      //-----------------------------------------
71      // Liberar memoria
72      free(A);
73      free(B);
74      free(C);
75
76      //=========================================
77      // MPI [4]: final
78      MPI_Finalize();
79
80      return 0;
81  }
82
83  //ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
84  // Vector de numeros aleatorios entre 0 y 100
85
86  double *generateRandomArray(int N){
87      double *x;
88      srand(time(NULL));
89      x = (double *)malloc(N*sizeof(double));
90      for (int i = 0; i < N; i++){
91          x[i] = rand() %100 + 1;
92          //printf("%f\n",x[i]);
93      }
94      return x;
95  }
```

Aunque el programa parece largo y complicado de entender, con una inspección rápida podemos observar que los cambios importantes son mínimos a su versión en serie. La parte más importante radica en la división del número de elementos que le toca a cada dominio. De esta manera podemos determinar cuantos elementos le corresponde calcular a cada procesador. Las demás partes son simplemente un formato pre-establecido que todo programa en paralelo usando MPI debe de llevar como se describión en los conceptos básicos.

2.3.3. Suma de los elementos de un vector de dimensión N

Finalmente, en esta sección presentamos el tercer problema que consiste en recibir un vector con N elementos y realiza la suma de todos los elementos de dicho vector.

Notemos que el siguiente programa tiene una estructura muy parecida a la de la suma de dos vectores. Al igual que el problema anterior es necesario distribuir el número de operaciones entre el número de procesadores (procesos) deseados. La principal diferencia radica en que la solución de este problema es un único valor que depende de la información proporcionada por cada componente del vector.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

double *generateRandomArray(int numberOfElements);

int main()
{
    int N = 10000;
    int i;
    double suma, cpu_time_used;
    double *x;
    //==================
    // MPI [1]: definiciones
    int numtasks, taskid;
    int ndims = 2;
    int dims[ndims];
    int periodicite[ndims];
    int reorganisation;
    int commID;
    int Nx, Ndom, iIni, iFin;
    double sumaglob;
    //==================

    //-----------------------------------------
    // Vector de N elementos
    //x = generateRandomArray(N); /* Opcion 1: Arbitrario */

    //=========================================
    // MPI [2]: inicio
    //-------------------------
    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    //-------------------------
    //Topologia (cartesiana)
    dims[0] = numtasks;
    dims[1] = 1;
    reorganisation = 0;
    periodicite[0] = 0;
    periodicite[1] = 0;
    MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodicite, reorganisation, &commID);
    //=========================================

    //=========================================
    // MPI [3]: Division del trabajo
    Ndom = (int)(1.0*N/numtasks);
    if (numtasks == 1) {
        Nx = N;
    }
    else {
        if(taskid==numtasks-1){
            Nx = N - Ndom*(numtasks-1);
        }
        else {
            Nx = Ndom;
        }
    }
    iIni = taskid*Ndom;
    iFin = iIni + Nx;
    //=========================================

    //-----------------------------------------
    // Operaciones: sumar sus componentes
    clock_t start, end;
    start = clock();
    //-------------------------
    suma = 0.0;
    for (i = iIni; i < iFin; i++) {
        suma = suma + x[i];
    }
    //-------------------------
    end = clock();
    cpu_time_used = ((double)(end-start))/CLOCKS_PER_SEC;
```

```
77
78      //=========================================
79      // MPI [4]: sumar componentes de cada procesador
80      MPI_Allreduce(&suma,&sumaglob,1,MPI_DOUBLE_PRECISION,MPI_SUM,comm1D);
81      //=========================================
82
83      //-----------------------------------------
84      // Imprimir
85      printf("Procesor %d de %d: \n \
86              Nx    = %d  \n \
87              iIni  = %d  \n \
88              iFin  = %d  \n \
89              suma  = %f  \n \
90              sumaT = %f  \n \
91              time  = %lf \n \n",taskid,numtasks,Nx,iIni,iFin,suma,sumaglob,cpu_time_used);
92
93      //-----------------------------------------
94      // Liberar memoria
95      free(x);
96
97      //=========================================
98      // MPI [5]: final
99      MPI_Finalize();
100     //=========================================
101
102     return 0;
103 }
```

Por lo tanto este problema requiere alguna clase de comunicación entre los procesos. Uno que sume todos los valores obtenidos de cada sub-dominio. Esto se hace mediante el comando `MPI_Allreduce` con `MPI_SUM`. El primero indica que la comunicación será de *todos contra todos* , es decir la información pedida la conocerán todos los procesos. El segundo es el comando asignado por MPI para indicar que lo que se desea hacer es sumar.

Existes otros detalles en este programa tales como la forma en que se dividió el dominio original, la topología, que se explicarán en el curso.

3. Algoritmos en paralelo usando GPUs: CUDA

La arquitectura unificada de dispositivos de cómputo (CUDA por sus siglas en inglés), es una plataforma de cómputo paralelo que contiene un conjunto de instrucciones, utilizadas para paralelizar algoritmos y ejecutarlos en una tarjeta gráfica o unidad de procesamiento gráfico (GPU).

Hoy en día, la GPU además de utilizarse para videojuegos, también se utiliza para hacer cómputo de propósito general. Se pueden resolver problemas en diferentes áreas como finanzas, gráficos por computadora, procesamiento de imágenes y video, algebra lineal, física, química, biología, etc. La arquitectura de la GPU contiene múltiples transistores en su unidad aritmética lógica, lo cual permite acelerar los algoritmos cuando se procesan una gran cantidad de datos controlados por una simple instrucción en paralelo (modelo SIMD) [3, 4, 5, 6].

3.1. Conceptos básicos de CUDA

En el curso veremos el modelo tradicional de programación en CUDA, el cual se resume en los siguientes pasos:

- Crear memoria en la GPU.
- Copiar memoria del CPU a la GPU.
- Ejecutar la función Kernel.
- Copiar memoria de la GPU a la CPU.

Imaginemos que tenemos nuestros datos en la CPU, a los cuales les queremos realizar un procesamiento en la GPU. Primero, debemos reservar memoria en la GPU. Luego, copiamos la memoria de la CPU a la GPU de los datos que queremos procesar. Después, ejecutamos la función que procesará los datos en la GPU, a esta función le llamaremos función Kernel, la cual devolverá resultados en la memoria de la GPU. Finalmente, debemos copiar la memoria en donde tenemos nuestros resultados, de la GPU a la memoria de la CPU.
La memoria de la GPU se divide en:

- Memoria global.
- Memoria constante.
- Memoria de textura.
- Memoria local y registros.
- Memoria compartida.

La memoria global, constante y de textura de la GPU son las que podemos usar para almacenar los datos que se copian desde la CPU. A diferencia de la memoria global que puede ser de lectura o escritura, la memoria constante y de textura solo son de lectura. La memoria local y registros se utilizan para mantener las variables que tienen un alcance local a los hilos dentro de la función Kernel. La memoria compartida se encuentra en el chip, por lo que el acceso a ella es más rápido que el acceso a la memoria global, y es la que mantiene los datos que son utilizados con mucha frecuencia para cada bloque de hilos.

3.2. Instalación

Para instalar CUDA, hay que seguir las instrucciones de la página oficial de CUDA:

https://developer.nvidia.com/cuda-toolkit

En el caso del sistema operativo Windows, podemos utilizar el ambiente de programación Visual Studio (versión Comunidad).

3.3. Ejemplos CUDA

A continuación, se muestran tres ejemplos para dar una rápida introducción al cómputo en paralelo usando CUDA.

3.3.1. Mi primer programa en paralelo usando CUDA

El primer ejemplo, consta de un archivo con extensión *.cu*, en el cual tenemos una función Kernel llamada *helloCUDA*, que imprime el índice del hilo y el índice el bloque que se están ejecutando, además se imprime una variable que se ha pasado como parámetro a la función.

```
1  Ejemplo 1: Hola Mundo (archivo HelloCUDA.cu)
2  #include <stdio.h>
3  #include <cuda_runtime.h>
4
5  __global__ void helloCUDA(float e){
6      printf("Soy el hilo %d del bloque %d con valor e=%f\n",threadIdx.x,blockIdx.x,e);
7  }
8
9  int main(void){
10     printf("\nHello World\n");
11
12     helloCUDA<<<3,4>>>(2.5f);
13
14     return(0);
15 }
```

En el cuerpo principal del programa, vemos la forma de llamar a una función Kernel, básicamente, debemos configurar dentro de los delimitadores ⋘`Dg,Db`⋙ la cantidad `Dg` de bloques de hilos y la cantidad `Db` de hilos dentro de cada bloque que queremos ejecutar, para que realicen lo que se encuentra dentro de la función Kernel.

Para compilar el programa se utiliza el compilador nvcc de CUDA de la siguiente manera:

`nvcc HelloCUDA.cu -o run_HelloCUDA.exe` → Windows
`nvcc HelloCUDA.cu -o run_HelloCUDA` → Linux

3.3.2. Suma de dos vectores de dimensión N

El segundo ejemplo consta de un archivo *Suma_Vectores.cu*, en donde podemos observar los pasos del modelo tradicional de programación en CUDA, para la implementación paralela de la suma de dos vectores inicializados con valores aleatorios.

En la función Kernel vemos que no es necesario crear un ciclo for para recorrer cada uno de los elementos de ambos vectores, en su lugar, solo necesitamos calcular el índice`idx` de las localidades de los vectores. Note que el índice `idx` solo depende de las variables `blockIdx`, `blockDim` y `threadIdx`, las cuales no han sido declaradas dentro de la función Kernel, estas variables se reconocen de forma automática y sus correspondientes valores dependen de la configuración elegida en ⋘ `Dg, Db` ⋙.

En este ejemplo estamos fijando el tamaño de los vectores $N = 24$ y el tamaño de los bloques de hilos `block_size =8`, entonces vamos a lanzar ⋘ 3, 8 ⋙ bloques de 8 hilos cada uno. Si elegimos una N que no sea múltiplo del tamaño de los bloques, por ejemplo, $N = 27$, entonces lanzaríamos ⋘ 4, 8 ⋙ 4 bloques de 8 hilos cada uno, que nos daría un total de 32 hilos lanzados, sin embargo, queremos que solo 27 hilos hagan el trabajo de sumar los elementos de ambos vectores, por este motivo es que ponemos la condición `if (idx <N)`, así solo los hilos con índice dentro del tamaño de los vectores pueden realizar la suma.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda_runtime.h>
4
5  // Función Kernel que se ejecuta en el Device.
6  __global__ void Suma_vectores(float *c_d,float *a_d,float *b_d, int N)
7  {
8    int idx = blockIdx.x * blockDim.x + threadIdx.x;
9    if (idx<N){
10       c_d[idx] = a_d[idx] + b_d[idx];
11   }
12 }
13
14 // Código principal que se ejecuta en el Host
15 int main(void){
16     float *a_h,*b_h,*c_h; //Punteros a arreglos en el Host
17     float *a_d,*b_d,*c_d;  //Punteros a arreglos en el Device
18     const int N = 24;  //Número de elementos en los arreglos  (probar 1000000)
19
20     size_t size=N * sizeof(float);
21
22     a_h = (float *)malloc(size); // Pedimos memoria en el Host
23     b_h = (float *)malloc(size);
24     c_h = (float *)malloc(size);//También se puede con cudaMallocHost
25
26     //Inicializamos los arreglos a,b en el Host
27     srand(time(NULL));
28     for (int i=0; i<N; i++){
29         //a_h[i] = (float)i;b_h[i] = (float)(i+1);
30         a_h[i] = rand() %100 + 1.0;
31         b_h[i] = rand() %100 + 1.0;
32     }
33
```

```
34      printf("\nArreglo a:\n");
35      for (int i=0; i<N; i++) printf("%f ", a_h[i]);
36      printf("\n\nArreglo b:\n");
37      for (int i=0; i<N; i++) printf("%f ", b_h[i]);
38
39      //Pedimos memoria en el Device
40      cudaMalloc((void **) &a_d,size);
41      cudaMalloc((void **) &b_d,size);
42      cudaMalloc((void **) &c_d,size);
43
44      //Pasamos los arreglos a y b del Host al Device
45      cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
46      cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
47
48      //Realizamos el cálculo en el Device
49      int block_size =8;
50      int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
51
52      Suma_vectores <<< n_blocks, block_size >>> (c_d,a_d,b_d,N);
53
54      //Pasamos el resultado del Device al Host
55      cudaMemcpy(c_h, c_d, size,cudaMemcpyDeviceToHost);
56
57      //Resultado
58      printf("\n\nArreglo c:\n");
59      for (int i=0; i<N; i++) printf("%f ", c_h[i]);
60
61      printf("\n\nFin del programa...\n");
62
63      // Liberamos la memoria del Host
64      free(a_h);
65      free(b_h);
66      free(c_h);
67
68      // Liberamos la memoria del Device
69      cudaFree(a_d);
70      cudaFree(b_d);
71      cudaFree(c_d);
72      return(0);
73  }
```

3.3.3. Suma de los elementos de un vector de dimensión N

Nuestro último ejemplo corresponde a la implementación del código serial y paralelo usando CUDA para sumar los elementos de un arreglo en punto flotante (double).

A continuación se presenta el código correspondiente a la suma de los elementos de un arreglo (archivos *sumatoria_main.cpp*, *kernel.h* y *kernel.cu*). Notar que en la función Kernel de este ejemplo podemos observar el uso de memoria compartida (_shared_) y de la función atómica `atomicAdd()`.

```
1   #include "kernel.h"
2
3   #define TPB 1024
4   #define ATOMIC 1 // 0 para no usar el atomicAdd
5
6   __global__ void sumOfArrayKernel(double *d_sum_total, double *d_A, long int n) {
7       const long int idx = threadIdx.x + blockDim.x * blockIdx.x;
8       const int s_idx = threadIdx.x;
9       __shared__ double s_data[TPB];
10
11      s_data[s_idx] = (idx<n) ? d_A[idx] : 0.0;
12      __syncthreads();
13
14      if (s_idx==0) {
15          double blockSum = 0.0;
16          for (int j = 0; j < blockDim.x; j++) {
17              blockSum += s_data[j];
18          }
19          //printf("Block_%d, blockSum = %f\n", blockIdx.x, blockSum);
20          if (ATOMIC) {
21              atomicAdd(d_sum_total, blockSum);
```

```
22          }
23          else {
24              *d_sum_total += blockSum; //Resultados no esperados
25          }
26      }
27  }
28
29  double sumOfArrayGPU(double *A, long int n){
30      double *d_A;
31      double *d_sum_total;
32      double sum_total;
33
34      //1. Crear memoria en la GPU
35      cudaMalloc(&d_sum_total, sizeof(double));
36      cudaMalloc(&d_A, n * sizeof(double));
37
38      //Inicializamos en cero
39      cudaMemset(d_sum_total, 0, sizeof(double));
40
41      //2. Copiar memoria (CPU->GPU)
42      cudaMemcpy(d_A, A, n * sizeof(double), cudaMemcpyHostToDevice);
43
44      //3. Ejecutar funciÛn Kernel
45      sumOfArrayKernel <<<(n+TPB-1)/TPB,TPB >>> (d_sum_total,d_A,n);
46      //sumOfArrayKernel_V2 << <(n + TPB - 1) / TPB, TPB >> > (d_sum_total, d_A, n);
47
48      //4. Copiar memoria (GPU->CPU)
49      cudaMemcpy(&sum_total, d_sum_total, sizeof(double), cudaMemcpyDeviceToHost);
50
51      cudaFree(d_sum_total);
52      cudaFree(d_A);
53      cudaDeviceReset();
54      return(sum_total);
55  }
```

La idea en esta implementación paralela es que cada bloque de hilos realice una suma parcial. La cantidad de elementos n del arreglo se divide entre el número de hilos por bloque (TPB). Luego, a cada bloque se le asigna un arreglo en la memoria compartida igual al tamaño del TPB, en donde se almacenan los elementos correspondientes del arreglo `d_A` que se encentra en memoria global. Después, uno de los hilos de cada bloque, realiza la suma parcial correspondiente, almacenando el resultado en la variable blockSum. Finalmente, las sumas parciales se van agregando en una sola variable `d_sum_total`, la cual guarda el resultado final de la sumatoria.

Para compilar el programa se utiliza el compilador `nvcc` de CUDA de la siguiente manera:

```
nvcc -arch=compute_60 kernel.cu sumatoria_main.cpp -o run_sumatoria
```

Finalmente, en el curso veremos algunas consideraciones importantes y al menos una mejora en la implementación de esta función Kernel. Además, veremos algunos ejemplos de procesamiento de imágenes.

Referencias

[1] Chandra, Rohit and Dagum, Leo and Kohr, David and Menon, Ramesh and Maydan, Dror and McDonald, Jeff (2001). Parallel programming in OpenMP, Morgan kaufmann.

[2] William Gropp and Ewing Lusk and Anthony Skjellum (1999). Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Scientific and Engineering Computation, Second edition.

[3] Cook, S. (2012). CUDA programming: a developer's guide to parallel computing with GPUs. Newnes

[4] Wilt, N. (2013). The cuda handbook: A comprehensive guide to gpu programming. Pearson Education.

[5] Cheng J., Grossman M. , and McKercher T. (2014). Professional CUDA C Programming, John Wiley and sons Inc.

[6] Storti, D., and Yurtoglu, M. (2015). CUDA for engineers: an introduction to high-performance parallel computing. Addison-Wesley Professional.