Práctica 4 – Suma de vectores

Fecha: 22/04/2022

Grupo: 4CDV1

Equipo: NetPower

Integrantes:

Alcibar Zubillaga Julián
De Luna Ocampo Yanina
Salinas Velazquez Jacob

Objetivos

Parte 1: Con base en el código de la práctica 3, donde se suman dos escalares en el GPU, modificarlo para que realice una suma de vectores paralelos.

Escenario

Los hilos (threads) en CUDA constituyen la base de la ejecución de aplicaciones en paralelo. Cuando se lanza un kernel se crea una malla de hilos donde todos ellos ejecutan el mismo programa. Es decir, el kernel especifica las instrucciones que van a ser ejecutadas por cada hilo individual. En la práctica anterior no hicimos uso de esta propiedad ya que lanzamos lo que podríamos llamar un “kernel escalar”, esto es, un bloque con un solo hilo de ejecución.

En esta práctica se va a hacer uso del paralelismo a nivel de hilo que proporciona CUDA lanzando varios hilos.

Nota: Se utilizará un servidor externo para realizar la práctica. Es necesario contar con un programa cliente ssh y credenciales de acceso al servidor externo.

Recursos necesarios para realizar la práctica

- 1 computadora personal
- Acceso a internet
- 1 programa cliente SSH* (Putty: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html)

INTRODUCCIÓN

¿Qué es CUDA?

CUDA (Arquitectura unificada de dispositivos de cómputo) es una arquitectura de procesamiento en paralelo creada por la empresa especializada en tarjetas gráficas Nvidia, en la que se trata de aprovechar la capacidad y potencia de sus GPU (Unidad de procesamiento gráfico) como una alternativa al procesamiento en CPU's tradicionales.

CUDA es una tecnología basada tanto en hardware como en software. En hardware al hacer que la arquitectura de sus tarjetas gráficas estén preparadas para poder explotar el paralelismo al contar con varios conjuntos de procesadores que reciben una misma instrucción para un vasto conjunto de datos. Entre ellas encontramos las arquitecturas Maxwell, Pascal entre otras.

En cuanto a software, CUDA provee una plataforma en forma de librerías y compiladores para que se puedan escribir programas que aprovechen el hardware de las GPUs dándole total control al desarrollador. [1]

¿Por qué CUDA?

Existen diferentes arquitecturas de computación paralela. Estás tienen su máximos exponentes en especificaciones y estándares tan potentes y utilizados como OpenMP o bien MPI los cuales en la actualidad son los que suelen ser utilizados en la mayoría de centros de investigación.

Lo que trata de hacer CUDA es desviarse de lo que buscan los procesadores tradicionales como los de la serie Xenon de Intel, en donde sus núcleos son muchísimo más rápidos que un núcleo de una GPU estándar (hasta 4.5 Ghz vs 1096 – 1020 Mhz) en vez de enfocarse en la rapidez de los núcleos, se enfoca en la cantidad de estos núcleos. Así, en vez de tener 24 o 36 núcleos a una velocidad de 4.5Ghz, una GPU cuenta con 640, 1020 y hasta 1280 núcleo (en las tarjetas de video serie titán) con una frecuencia de reloj de 1078Mhz.

Esto permite aprovechar la técnica SIMD de simple instrucción para múltiple data. Por esa razón, CUDA se acopla demasiado bien para aplicar computación paralela y así maximizar el rendimiento de las aplicaciones.

Cabe mencionar que CUDA es transparente al sistema operativo, es decir que CUDA ofrece soporte para sistemas operativos Windows, GNU/Linux y para sistemas operativos MacOs. Su rendimiento no depende del sistema operativo en sí, si no en las condiciones de trabajo que se encuentre la máquina o clúster sobre el cual se hará funcionar la aplicación. [2]

Paralelizar una aplicación a través de CUDA

Para comprobar, analizar y determinar el mejoramiento real de usar computación paralela en un problema real (hay que tener en cuenta que no todos los problemas se pueden paralelizar ya que por lo general son inherentemente secuenciales) se utilizarán una aplicación llamada ScanSky, desarrollada y escrita por la Dra. Ana Moretón Fernández, el Dr. Javier Fresno y el Dr. Arturo González-Escribano del grupo de investigación Trasgo, de la escuela superior de ingeniería informática de la Universidad de Valladolid, España, como parte de la práctica número tres del curso de computación paralela.

Esta aplicación recibe como entrada un archivo que consiste en números separados por un salto de línea. Estos números representan el color de cada uno de los píxeles de una imagen. El objetivo de esta aplicación es determinar con precisión el número de cuerpos celestes presentes en dicha imagen. [3]

Parte 1: Correr y analizar el programa 04_memo.cu

Utiliza un cliente SSH para conectarte al servidor lambda 01.

(te puedes conectar al servidor lambda de la misma forma que lo hiciste en la práctica 01).

Dentro del servidor Lambda. Acceder a la carpeta “hpc_4AV1

Accedemos a nuestra carpeta de trabajo y hacemos una copia del programa 05_suma.cu a un nuevo archivo llamado 06_sumavect.cu, ejecuta el siguiente comando:

/hpc_4AV1/estudiantes/<tu_carpeta_de_trabajo>/u01/# cp 05_suma.cu 06_sumavect.cu

<tu_carpeta_de_trabajo> es el nombre de tu carpeta de trabajo. Este comando realizará una copia del programa “05_suma.cu” en un archivo llamado 06_sumavect.cu

Modifica las siguientes opciones:

1 Declaración de variables

Para hacer más fácil la suma de vectores y el paralelismo, se trabajará con punteros. Declara las siguientes variables:

```
int *s1, *s2, *suma; //Generar los vectores y el resultado de la suma en el CPU
int *s1_G, s2_G, *suma_G; //Vectores y su suma en el GPU
int *suma_H; //Suma copiada del GPU al CPU
```

2 Reservar espacio en memoria

Hay que reservar espacio tanto en el CPU como en el GPU

- Reserva de memoria en el CPU:

```
s1     = (int *)   malloc(sizeof(int) * N);
s2     = (int *)   malloc(sizeof(int) * N);
suma   = (int *)   malloc(sizeof(int) * N);
suma_H = (int *)   malloc(sizeof(int) * N);
```

- Reserva de memoria en el GPU:

```
cudaMalloc( (void **)&s1_G,   sizeof(int) * N);
cudaMalloc( (void **)&s2_G,   sizeof(int) * N);
cudaMalloc( (void **)&suma_G, sizeof(int) * N);
```

3 Rellenar los vectores con valores aleatorios

Los más conveniente es crear una función que rellene el arreglo con valores aleatorios.

```
void rellena(int *arreglo){
int i;
for(i=0; i<N; i++){
    arreglo[i] = rand()%100;
  }
}
```

4 Suma en el CPU

Realizaremos una función que sume los vectores en el CPU

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

5 Suma en el GPU

La función que suma los dos vectores en el GPU será muy parecida a la suma que hicimos. Haremos que se sume cada elemento de los arreglos correspondientes dentro de cada *thread*. Para identificar el *thread* específico dentro de cada arreglo se usará la expresión: **threadIdx.x + (blockIdx.x * blockDim.x);**

```
__global__ void sumaGPU(int* a, int* b, int* c)
{
  int id;
  id = threadIdx.x + (blockIdx.x*blockDim.x);

  c[id] = a[id] + b[id];

}
```

5 Copiar los vectores del CPU al GPU

Para copiar los vectores del CPU al GPU se usarán las siguientes expresiones:

```
cudaMemcpy(s1_G,s1, sizeof(int) * N, cudaMemcpyHostToDevice);
cudaMemcpy(s2_G,s2, sizeof(int) * N, cudaMemcpyHostToDevice);
```

6 Calcular el número de *threads* y bloques

Con base en N podemos calcular el tamaño del bloque y el número de threads (en este caso serán 1000):

```
procesos = 1000;
bloque   = N / procesos;
```

Llamar el kernel

Finalmente, llamaremos el kernel del GPU usando la expresión:

```
sumaGPU<<<bloque,procesos>>>(s1_G,s2_G,suma_G);
```

7 Imprime el resultado

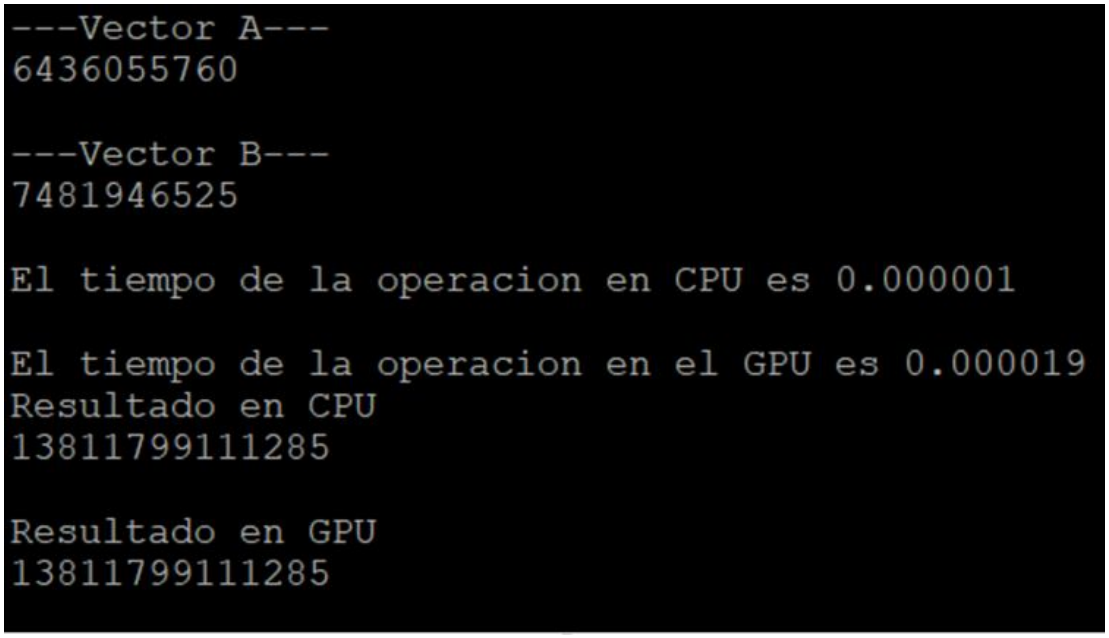
Finalmente realiza una función que imprima el resultado de la suma en el GPU y en el CPU.

Compila y ejecuta el programa, usa los siguientes comandos:

nvcc 05_sumavect.cu -o 06_sumavect.x

./06_sumavect.x

INCLUYE AQUÍ LA CAPTURA DE PANTALLA CON EL RESULTADO DEL PROGRAMA EJECUTÁNDOSE



Captura de pantalla con la ejecución del programa

Incluye aquí el código completo:

```
//Inicio del código
  GNU nano 4.8                                                    06_sumavect.cu
// includes
#include <stdio.h>
#include <cassert>
#include <stdlib.h>
#include <ctime>
#include <time.h>
#include <cuda_runtime.h>
#define N 10

__global__ void sumaGPU(int* a, int* b, int* c)
{
  int id;
  id = threadIdx.x + (blockIdx.x * blockDim.x);
  c[id] = a[id] + b[id];


}

void sumaCPU(int* a, int* b, int *c)
{
   int i;

   for(i=0; i<N; i++)
   {
       c[i] = a[i] + b[i];
   }

}

void rellena(int* arreglo){
   int i;
   for(i = 0;i < N;i++){
        arreglo[i] = rand()%10;
   }
}

void imprime(int* arreglo){
   for(int i = 0; i < N; i++){
        printf("%d",arreglo[i]);
   }
   printf("\n");
}
```

```
int comparar(int* a, int* b){
  int i;
  for(i = 0; i < N; i++){assert(a[i] == b[i]);}
}

// MAIN: rutina principal ejecutada en el  CPU
int main(int argc, char** argv)
{
// declaracion
unsigned t0, t1;
int bloque, procesos;
int* s1, *s2, *suma_CPU, *s1_g, *s2_g, *suma_g, *suma_h;

//reserva memoria CPU
s1 = (int*) malloc(sizeof(int)*N);
s2 = (int*)malloc(sizeof(int)*N);
suma_CPU = (int*)malloc(sizeof(int)*N);
suma_h = (int*)malloc(sizeof(int)*N);
// reserva en el GPU
cudaMalloc( (void**)&s1_g, sizeof(int)*N);
cudaMalloc((void**)&s2_g, sizeof(int)*N);
cudaMalloc((void**)&suma_g, sizeof(int)*N);
//rellenamos los vectores
srand(time(0));
rellena(s1);
rellena(s2);
printf("\n---Vector A---\n");
imprime(s1);
printf("\n---Vector B---\n");
imprime(s2);
//Llamamos a la funcion en el CPU
t0 = clock();
sumaCPU(s1,s2,suma_CPU);
t1 = clock();
printf("\nEl tiempo de la operacion en CPU es %f\n", double(t1-t0)/CLOCKS_PER_SEC);
//enviar vectores al GPU
cudaMemcpy(s1_g,s1, sizeof(int) * N,cudaMemcpyHostToDevice);
cudaMemcpy(s2_g,s2, sizeof(int) * N,cudaMemcpyHostToDevice);


//Llamamos a la función en el GPU
procesos = 10;
bloque = N/procesos;
```

```
//Llamamos a la función en el GPU
procesos = 10;
bloque = N/procesos;
t0 = clock();
sumaGPU<<<bloque,procesos>>>(s1_g,s2_g,suma_g);
t1 = clock();
printf("\nEl tiempo de la operacion en el GPU es %f\n",double(t1-t0)/CLOCKS_PER_SEC);

//copiamos el resultado de suma
cudaMemcpy(suma_h,suma_g, sizeof(int) * N,cudaMemcpyDeviceToHost);

//imprime resultado
printf("Resultado en CPU\n");
imprime(suma_CPU);
printf("\n");
printf("Resultado en GPU\n");
imprime(suma_h);

comparar(suma_h,suma_CPU);

cudaFree(s1_g);
cudaFree(s2_g);
cudaFree(suma_g);

printf("\npulsa INTRO para finalizar...");
fflush(stdin);

char tecla = getchar();
return 0;

}
```

```
//Fin del código
```

Parte 2: Cuestionario

¿Podrías paralelizar la función para rellenar los arreglos?

Si, mediante Funciones

De la pregunta anterior ¿Cómo lo harías (puedes incluir código)?

```
int A[N],B(N), i;

char RINL;

for (1=0 ; i<N it+);

printf ("\nIngresar los valores en posicion &d -> ". i);

scanf ("%d %d", &A[il, &B[il);

for (i=0 ; i<N : i++);

if (Ali]›Bli])

R[i]z'A'; it+)

else

if (Ali‹Bli]) Rlil='B';

else Rli]=":

printf("\n A B R");

for (i=0 : i<N ; ¡t+)

printf("\n &3d &3d", Alil,Bli]);


__global__void rellenaGPU( ){
        int id = blockIdx.x *
blockDim.x + threadIdx.x;
        c[id] = rand( ) % 100;
}
```

¿Podrías paralelizar la función para mostrar el resultado de la suma?

Si, de una manera muy similar a la pregunta anterior

De la pregunta anterior ¿Cómo lo harías (puedes incluir código)?

```
{
      for (i=0;i< num_steps; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
   }
   Suma = step * sum;
   printf("%e \n",Suma);
}
__global__void imprimeGPU( ){
            int id = blockIdx.x *
blockDim.x + threadIdx.x;
            printf(“%d”, c[id]);
}
```

Conclusiones

Podemos concluir que la realización de esta práctica nos permite acercarnos de una manera totalmente eficaz a lo que es la paralización y su implementación dentro de un servidor, observamos la estructura de un programa e ideamos las respectivas funciones para el cometido de la práctica, haciendo que quede muy claro el manejo de esto y prepararnos para su uso en proximas practicas.

Bibliografía

1. *Computación paralela a través de CUDA | Decimocuarta Edición - ECYS*. (s. f.). https://revistaecys.github.io/14Edicion/05-etejaxun.html
2. *CUDA paralelo dinámico - programador clic*. (s. f.). programador clic. https://programmerclick.com/article/4887420488/
3. *¿CUDA? Programación Paralela de GPU – NVIDIA - MR Solutions*. (s. f.). MR Solutions. https://www.mrsolutions.com.mx/cuda-programacion-paralela-de-gpu-nvidia/

Consideraciones finales

Descarga el documento antes de llenarlo.

Este documento se debe llenar en equipo, aunque la práctica la deben hacer TODOS los integrantes de este.

Después de llenar el documento, guárdalo como PDF y envíalo a través de la plataforma TEAMS, en la pestaña de tareas correspondiente. Solamente lo tiene que subir uno de los integrantes. Pero deben incluir TODOS los nombres de los integrantes del equipo en la primera página.

Queda estrictamente prohibido cualquier tipo de plagio a otros equipos o grupos. En caso de que ocurra, se anulará la práctica y se descontarán dos puntos a los equipos involucrados.