Laboratorio no. 5 Programación Paralela

author: Marco Jurado 20308

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:33:58_PDT_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
```

```
Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-p662y5w8
  Running command git clone --filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-p662y5w8
  Resolved https://github.com/andreinechaev/nvcc4jupyter.git to commit 0a71d56e5dce3ff1f0dd2c47c29367629262f527
  Preparing metadata (setup.py) ... done
```

```
%load_ext nvcc_plugin
```

```
The nvcc_plugin extension is already loaded. To reload it, use:
  %reload_ext nvcc_plugin
```

```
!pip install pycuda
```

```
Requirement already satisfied: pycuda in /usr/local/lib/python3.10/dist-packages (2022.2.2)
Requirement already satisfied: pytools>=2011.2 in /usr/local/lib/python3.10/dist-packages (from pycuda) (2023.1.1)
Requirement already satisfied: appdirs>=1.4.0 in /usr/local/lib/python3.10/dist-packages (from pycuda) (1.4.4)
Requirement already satisfied: mako in /usr/local/lib/python3.10/dist-packages (from pycuda) (1.3.0)
Requirement already satisfied: platformdirs>=2.2.0 in /usr/local/lib/python3.10/dist-packages (from pytools>=2011.2->pycuda) (3.11.0)
Requirement already satisfied: typing-extensions>=4.0 in /usr/local/lib/python3.10/dist-packages (from pytools>=2011.2->pycuda) (4.5.0)
Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.10/dist-packages (from mako->pycuda) (2.1.3)
```

```
import pycuda.driver as drv
import pycuda.autoinit
drv.init()
print("%d device(s) found." % drv.Device.count())
for i in range(drv.Device.count()):
  dev = drv.Device(i)
  print("Device #%d: %s" % (i, dev.name()))
  print(" Compute Capability: %d.%d" % dev.compute_capability())
  print(" Total Memory: %s GB" % (dev.total_memory() // (1024 * 1024 * 1024)))
```

```
1 device(s) found.
Device #0: Tesla T4
 Compute Capability: 7.5
 Total Memory: 14 GB
```

Ahora que el ambiente esta listo para ser ejecutado con CUDA procedemos a realizar los ejercicios de esta hoja de trabajo.

```
%%cuda --name lab5.cu

#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>


__global__ void vectorAdd(float *a, float *b, float *c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
        printf("Thread no. %d: %.2f + %.2f = %.2f\n", idx, a[idx], b[idx], c[idx]);
    }
}

int main() {
    int n = 500;  // Tamaño de los vectores
```

```
    float *h_a, *h_b, *h_c; // Vectores en la CPU
    float *d_a, *d_b, *d_c; // Vectores en la GPU
    size_t size = n * sizeof(float);

    // Alojar memoria en la CPU
    h_a = (float *)malloc(size);
    h_b = (float *)malloc(size);
    h_c = (float *)malloc(size);

    if (h_a == nullptr || h_b == nullptr || h_c == nullptr) {
        std::cerr << "Error al alojar memoria en la CPU." << std::endl;
        return 1;
    }

    // Inicializar los vectores en la CPU con valores aleatorios
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        h_a[i] = static_cast<float>(rand()) / RAND_MAX;
        h_b[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    // Alojar memoria en la GPU
    cudaError_t cudaStatus;
    cudaStatus = cudaMalloc(&d_a, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_a: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMalloc(&d_b, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_b: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMalloc(&d_c, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_c: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Copiar datos desde la CPU a la GPU
    cudaStatus = cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la CPU a la GPU para d_a: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la CPU a la GPU para d_b: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Configuración de la cuadrícula y bloque
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    // Lanzar el kernel de CUDA
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);

    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al lanzar el kernel de CUDA: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Copiar el resultado de la GPU a la CPU
    cudaStatus = cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la GPU a la CPU para d_c: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Imprimir el vector resultante y la suma
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        std::cout << "Resultado[" << i << "]: " << h_c[i] << std::endl;
        sum += h_c[i];
    }
```

```
    std::cout << "Suma total: " << sum << std::endl;

    // Liberar memoria
    free(h_a);
    free(h_b);
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

```
'File written in /content/src/lab5.cu'
```

```
!nvcc -arch=sm_75 /content/src/lab5.cu -o "/content/src/lab5.o"
```

```
!chmod 755 /content/src/lab5.o
!/content/src/lab5.o
```

```
Resultado[443]: 1.52453
Resultado[444]: 0.731451
Resultado[445]: 1.59552
Resultado[446]: 0.940135
Resultado[447]: 1.58701
Resultado[448]: 1.55033
Resultado[449]: 0.490661
Resultado[450]: 1.08542
Resultado[451]: 1.56452
Resultado[452]: 0.81321
Resultado[453]: 0.677516
Resultado[454]: 0.62893
Resultado[455]: 0.446066
Resultado[456]: 1.01361
Resultado[457]: 0.335382
Resultado[458]: 1.38224
Resultado[459]: 1.69053
Resultado[460]: 0.927138
Resultado[461]: 1.4633
Resultado[462]: 0.580979
Resultado[463]: 1.23457
Resultado[464]: 1.05312
Resultado[465]: 0.782231
Resultado[466]: 1.321
Resultado[467]: 1.46485
Resultado[468]: 1.12484
Resultado[469]: 0.732442
Resultado[470]: 0.554185
Resultado[471]: 1.78548
Resultado[472]: 1.56701
Resultado[473]: 0.233901
Resultado[474]: 0.980771
Resultado[475]: 1.06294
Resultado[476]: 1.21048
Resultado[477]: 1.77518
Resultado[478]: 0.576546
Resultado[479]: 0.922764
Resultado[480]: 0.734987
Resultado[481]: 0.768206
Resultado[482]: 1.05175
Resultado[483]: 1.19019
Resultado[484]: 1.37543
Resultado[485]: 0.930361
Resultado[486]: 1.10114
Resultado[487]: 1.49741
Resultado[488]: 1.8367
Resultado[489]: 1.6174
Resultado[490]: 0.65574
Resultado[491]: 0.725377
Resultado[492]: 0.206576
Resultado[493]: 0.657669
Resultado[494]: 1.51031
Resultado[495]: 1.41279
Resultado[496]: 1.28543
Resultado[497]: 1.30128
Resultado[498]: 1.41406
Resultado[499]: 0.51377
Suma total: 505.721
```

Vamos a realizar el mismo código pero quitando los prints para poder ver el tiempo de ejecución del programa tal y como fue solicitado para los cálculos de speedup.

```
%%cuda --name lab5_timed.cu

#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>

__global__ void vectorAdd(float *a, float *b, float *c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

int main() {
    int n = 1000000;  // Tamaño de los vectores
    float *h_a, *h_b, *h_c; // Vectores en la CPU
    float *d_a, *d_b, *d_c; // Vectores en la GPU
    size_t size = n * sizeof(float);

    // Alojar memoria en la CPU
    h_a = (float *)malloc(size);
    h_b = (float *)malloc(size);
    h_c = (float *)malloc(size);

    if (h_a == nullptr || h_b == nullptr || h_c == nullptr) {
        std::cerr << "Error al alojar memoria en la CPU." << std::endl;
        return 1;
    }

    // Inicializar los vectores en la CPU con valores aleatorios
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        h_a[i] = static_cast<float>(rand()) / RAND_MAX;
        h_b[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    // Alojar memoria en la GPU
    cudaError_t cudaStatus;
    cudaStatus = cudaMalloc(&d_a, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_a: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMalloc(&d_b, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_b: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMalloc(&d_c, size);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al alojar memoria en la GPU para d_c: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Copiar datos desde la CPU a la GPU
    cudaStatus = cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la CPU a la GPU para d_a: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }
    cudaStatus = cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la CPU a la GPU para d_b: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Configuración de la cuadrícula y bloque
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
```

```
    // Lanzar el kernel de CUDA
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);

    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al lanzar el kernel de CUDA: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Copiar el resultado de la GPU a la CPU
    cudaStatus = cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        std::cerr << "Error al copiar datos desde la GPU a la CPU para d_c: " << cudaGetErrorString(cudaStatus) << std::endl;
        return 1;
    }

    // Imprimir el vector resultante y la suma
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += h_c[i];
    }
    std::cout << "Suma total: " << sum << std::endl;

    // Liberar memoria
    free(h_a);
    free(h_b);
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

```
'File written in /content/src/lab5_timed.cu'
```

```
!nvcc -arch=sm_75 /content/src/lab5_timed.cu -o "/content/src/lab5_timed.o"
```

```
%%time

!chmod 755 /content/src/lab5_timed.o
!/content/src/lab5_timed.o
```

```
Suma total: 999672
CPU times: user 13.9 ms, sys: 71 µs, total: 13.9 ms
Wall time: 412 ms
```

Ya que sabemos y hemos obtenido los valores para el codigo en secuencial con cuda vamos a diseñar el formato paralelo y correrlo para obtener el tiempo del mismo y hacer los respectivos calculos.

```
%%writefile secuencial.cpp

#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    int n = 1000000;  // Tamaño de los vectores
    float *h_a, *h_b, *h_c; // Vectores en la CPU
    size_t size = n * sizeof(float);

    // Alojar memoria en la CPU
    h_a = (float *)malloc(size);
    h_b = (float *)malloc(size);
    h_c = (float *)malloc(size);

    // Inicializar los vectores en la CPU con valores aleatorios
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        h_a[i] = static_cast<float>(rand()) / RAND_MAX;
        h_b[i] = static_cast<float>(rand()) / RAND_MAX;
    }
```

```
    // Realizar la suma de los vectores en forma secuencial
    for (int i = 0; i < n; i++) {
        h_c[i] = h_a[i] + h_b[i];
    }

    // Imprimir la suma total
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += h_c[i];
    }
    std::cout << "Suma total: " << sum << std::endl;

    // Liberar memoria
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
 Overwriting secuencial.cpp
```

```
%%bash

g++ secuencial.cpp -o secuencial
time ./secuencial
```

```
 Suma total: 999872

 real    0m0.047s
 user    0m0.042s
 sys     0m0.005s
```

```
import os

# Obtener la cantidad de núcleos (cores) disponibles
num_cores = os.cpu_count()

print("Número de núcleos (cores) disponibles:", num_cores)
```

```
 Número de núcleos (cores) disponibles: 2
```

Luego de que tenemos el tiempo de ejecucion de ambos programas procedemos a realizar los calculos de speedup y efficiency de nuestro programa paralelo. Para ello tomaremos un tamaño de 1,000,000 y realizamos la ejecución donde obtenemos lo siguiente:

- Tiempo secuencial: 47 ms
- Tiempo paralelo: 13.9 ms
- Speedup = 47 ms / 13.9 ms ≈ 3.38
- Efficiency = 3.38 / 2 ≈ 1.69