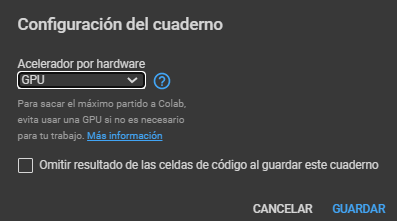
DESARROLLO DEL PROYECTO

**Entorno de trabajo**

El proyecto se realizó en Colab para lo cual se configura a entorno de trabajo en GPU y guardar



El lenguaje de programación que utilizaremos será c++, para que el Colab acepte este lenguaje se agrega y ejecuta los siguientes códigos

!apt-get --purge remove cuda nvidia\* libnvidia-\*

!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge

!apt-get remove cuda-\*

!apt autoremove

!apt-get update

!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local\_installers/cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64 -O cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64.deb

!dpkg -i cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64.deb

!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub

!apt-get update

!apt-get install cuda-9.2

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

%load\_ext nvcc\_plugin

Ejecutamos celda y agregamos código

!nvidia-smi

Igualmente ejecutamos esta celda.

**Especificaciones de código**

**1.%%cu**

¿por qué usar CUDA?

CUDA reúne varias cosas:

hardware masivo en paralelo diseñado para ejecutar código genérico (no gráfica), con controladores adecuados para hacerlo.

Un lenguaje de programación basado en C para programar dicho hardware y un lenguaje ensamblador que otros lenguajes de programación pueden usar como destino.

Un kit de desarrollo de software que incluye bibliotecas, varias herramientas de depuración, creación de perfiles y compilación, y enlaces que permiten que los lenguajes de programación del lado de la CPU invoquen el código del lado de la GPU.

El objetivo de CUDA es escribir código que pueda ejecutarse en arquitecturas SIMD paralelas masivas compatibles: esto incluye varios tipos de GPU además de hardware que no es GPU como nVidia Tesla. El hardware masivamente paralelo puede ejecutar un número significativamente mayor de operaciones por segundo que la CPU, a un costo financiero bastante similar, que ofrece mejoras de rendimiento de 50 × o más en situaciones que lo permiten.

Uno de los beneficios de CUDA sobre los métodos anteriores es que hay disponible un lenguaje de propósito general, en lugar de tener que usar pixel y sombreadores de vértices para emular las computadoras de propósito general. Ese lenguaje se basa en C con algunas palabras clave y conceptos adicionales, lo que hace que sea bastante fácil para los programadores que no usan GPU.

También es una señal de que NVIDIA está dispuesto a soportar la paralización de uso general en su hardware: ahora suena menos como "piratear con la GPU" y más como "usar una tecnología compatible con proveedores", y eso hace que adopción más fácil en presencia de partes interesadas no técnicas.

para empezar a utilizar CUDA, descargar el SDK, lea el manual (en serio, no es que complicado si ya conoce C) y comprar un hardware compatible con CUDA (que puede usar el emulador en un primer momento, pero el rendimiento siendo el último punto de esto, es mejor si realmente puedes probar tu código).

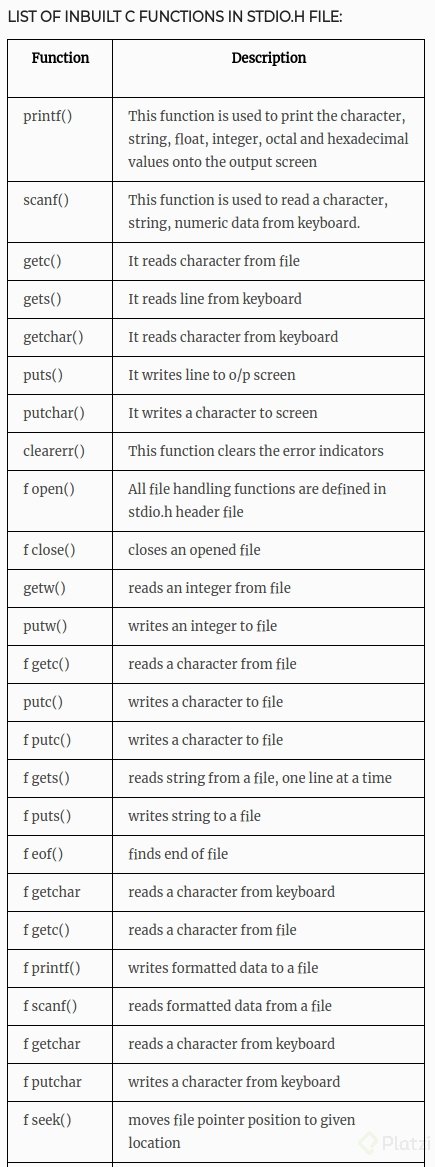
2. #include "cuda\_runtime.h"

cuda\_runtime\_api.h define los funciones y tipos de host públicos para la API CUDA tiempo de ejecución

cuda\_runtime.h define todo cuda\_runtime\_api.h hace, asi como de tipo incorporado definiciones y superposiciones de funciones para las extensiones de lenguaje CUDA y funciones intrínsecas dispositivo.

3. #include <stdio.h>

Importa / Incluye las funciones, constantes y macros desde la librería de Entrada / Salida estándar (standar input/output los periféricos estándar son, monitor, teclado, disco duro, impresora … y esta librería te permite leer desde o enviar información hacia estos periféricos como leer variables de ese teclado, o enviar un texto a un archivo en disco.



**4.** #include <stdlib.h>

(std-lib: standard library o biblioteca estándar). Es el archivo de cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras. Es compatible con C++ donde se conoce como cstdlib.



5. #include <cuda.h>

cuda.h define el host público funciones y tipos para la API de controlador CUDA

6. #define MAXVALUE 10000

Define el valor máximo = 10000

7.void numberGen(int N, int max, int\* store)

{

    int i;

    srand(time(0));

    for (i = 0; i < N; i++)

        store[i] = rand() % max;

}

Declarando la variable i, tiempo para generar números aleatorios almacena en el arreglo

7. \_\_global\_\_ void countOdds(int\* d, int N, int\* odds)

{

    extern \_\_shared\_\_ int count[];

    int myID = blockIdx.x \* blockDim.x + threadIdx.x;

    int localID = threadIdx.x;

    count[localID] = 0;

    if (myID < N)

        count[localID] = (d[myID] % 2);

    \_\_syncthreads();

    // fase de reducción

    int step = 1;

    while (((localID | step) < blockDim.x) && ((localID & step) == 0))

    {

        count[localID] += count[localID | step];

        step <<= 1;

        \_\_syncthreads();

    }

    // adiciona al contador global

    if (localID == 0)

        atomicAdd(odds, count[0]);

}

-Función \_\_global\_\_ que se puede llamar desde el host, y se ejecuta en el dispositivo

-Asignamos memoria compartida dinámicamente (extern significa que la matriz hace referencia a la memoria compartida declarada en otro lugar)

-Añadiendo el tamaño del Grid y del Bloque

-Ponemos el id en cada hilo

8. int sharedSize(int b)

{

    return b \* sizeof(int);

}

Genera Tamaño compartido y devuelve la cantidad de memoria asignada

9. int main(int argc, char\*\* argv)

{

    int N = 5000;

    int\* ha, \* hres, \* da, \* dres;   // punteros host (h\*) y device (d\*)

    ha = new int[N];

    hres = new int[1];

    cudaMalloc((void\*\*)&da, sizeof(int) \* N);

    cudaMalloc((void\*\*)&dres, sizeof(int) \* 1);

    numberGen(N, MAXVALUE, ha);

    cudaMemcpy(da, ha, sizeof(int) \* N, cudaMemcpyHostToDevice);

    cudaMemset(dres, 0, sizeof(int));

    int blockSize, gridSize;

    cudaOccupancyMaxPotentialBlockSizeVariableSMem(&gridSize, &blockSize, (void\*)countOdds, sharedSize, N);

    gridSize = ceil(1.0 \* N / blockSize);

    printf("Grid : %i    Block : %i\n", gridSize, blockSize);

    countOdds << < gridSize, blockSize, blockSize \* sizeof(int) >> > (da, N, dres);

    cudaMemcpy(hres, dres, sizeof(int), cudaMemcpyDeviceToHost);

   // chequeo

    int oc = 0;

    for (int i = 0; i < N; i++)

        if (ha[i] % 2){

            printf("%i ",ha[i]);

            oc++;

        }

    printf("\nCantidad de numeros impares - Chequeo\n");

    printf("%i %i\n", \*hres, oc);

    cudaFree((void\*)da);

    cudaFree((void\*)dres);

    delete[]ha;

    delete[]hres;

    cudaDeviceReset();

    getchar();

    return 0;

}

-Definimos la cantidad de numeros que se generaran aleatoriamente

-Define punteros host (h\*) y device (d\*)

-Declaramos un arreglo en la variable “ha”

-Declaramos un arreglo en la variable “hres”

- cudaMalloc: Almacena un espacio en la memoria

- cudaMemcpy : Copia datos a la memoria del dispositivo

- cudaFree((void\*)da); #Libera memoria de da

- cudaFree((void\*)dres);#Libera memoria de dres

- delete[]ha; #Operacion eliminar ha

- delete[]hres;#Operacion eliminar hres

- cudaDeviceReset(); #Libera contexto de cuda , quita las asiganciones de todo los dispositivos

**Páginas de ayuda**

**\*para Colab**

<https://www.marketing-branding.com/google-colaboratory-colab-guia-completa-espanol/>

**Bibliografía**

<https://stackoverrun.com/es/q/1269255>

<https://es.wikipedia.org/wiki/Stdlib.h#:~:text=h%20(std%2Dlib%3A%20standard,control%20de%20procesos%20y%20otras>.