UNIVERSIDAD AUTONOMA DE MADRID

Procesamiento de Grandes Volúmenes de Datos

Programación GPGPU y Computación Cuántica

Parte 1: Programación GPGPU

Se valorará hasta **3 puntos** en la nota de la Práctica 2.

0. Recursos de la GPU

En un nuevo cuaderno de Google Colab

Ir al enlace https://colab.research.google.com en un Navegador y haga Click en Nuevo Cuaderno

Seleccione la utilización de un coprocesador GPU

Click to Runtime > Change > Hardware Accelerator GPU .

Compruebe las características del sistema que le han proporcionado

```
!lscpu

!free -h
```

Verifique la versión de CUDA instalada

```
!nvcc --version


!nvidia-smi
```

Compruebe el directorio actual de trabajo

```
!pwd
!ls -la
!ls /
```

Compruebe las características de la GPU

```
%cd /usr/local/cuda/samples/1_Utilities/deviceQuery/
%ls
!make


!./deviceQuery
```

1. Suma de 2 vectores

El código de ejemplo en el tutorial suma de los elementos de un vector realiza la suma de dos vectores en la GPU.

1. Comente los diferentes casos propuestos en el ejemplo y conteste a las preguntas.

2. Suma de 2 matrices

Se propone extender el código del ejemplo anterior para que realice la resta (o suma) de matrices cuadradas de dimensión N.

- Configure adecuadamente el Grid de threads para aceptar matrices de cualquier tamaño.
- En el kernel, utilice las variables blockIdx y threadIdx adecuadamente para acceder a una estructura bidimensional.

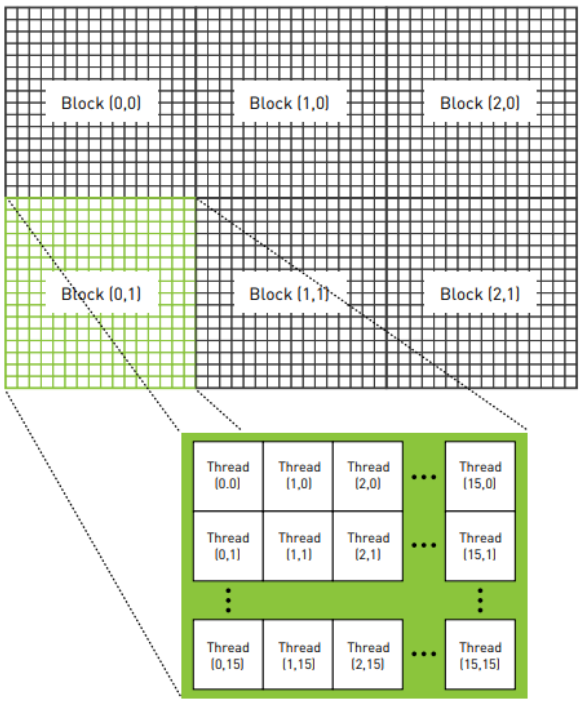


Figure 5.2 A 2D hierarchy of blocks and threads that could be used to process a 48 x 32 pixel image using one thread per pixel

Refs:

http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf

3. Stencil1d: Estudiar el efecto de la memoria compartida

Descripción:

El código stencil 1D es útil para entender los beneficios y uso de memoria compartida en una GPU. Para hacerlo explicito conviene valorar los resultados utilizando el generador de perfiles (nvprof) que muestra los cuellos de botella y su efecto en la aceleración final que se consigue.

Pasos:

1. Primero compile y ejecute el código sin usar memoria compartida. Hacer un perfil con nvprof y sacar el comportamiento temporal.

2. Use el generador de perfiles para determinar cuál es el problema.

3. Introduzca la modificación en el código para hacer uso de la memoria compartida.

Valore la situación que sucede cuando no se usa la función __syncthreads (). Ejecute el código varias veces y observe cuando se obtienen errores semi-aleatorios en la salida.

4. Añadiendo __syncthreads () evalue después de las diferentes modificaciones la aceleración obtenida.

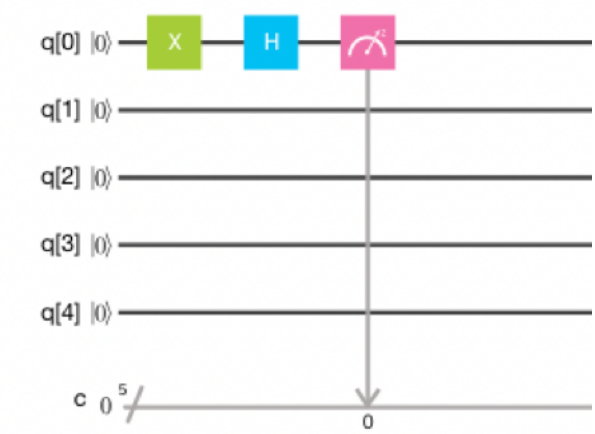
Parte 2 : Programación con QisKit: (Computación cuántica)

Se valorará hasta **3 puntos** en la nota de la Práctica 2.

5. Puertas Cuánticas

Puerta de Hadamard

La puerta de Hadamard crea una superposición básica 50-50. Por lo tanto, al medirla se obtendrá (teóricamente) un resultado de 0 o 1 con una probabilidad del 50% para cada uno de los estados.



La figura muestra un circuito básico construido en el entorno:

IBM Quantum Experience

https://quantum-computing.ibm.com

Se empieza con un qubit inicializado en “mínima energía”, |0>, y le aplicamos una puerta Hadamard (caja azul). A la salida de la puerta, el qubit está en superposición. Luego, lo medimos (caja rosa) y se toma el resultado de la medida que corresponde al estado en el que colapsa y por tanto ya se puede guardar como un bit clásico: 0 o 1.
La medida es necesario repetirla muchas veces (shots= 1024). El histograma debajo del circuito representa los resultados: Aproximadamente el 50% de las medidas proporcionaron un 0 y el 50% proporcionó un 1, como se esperaba. Tenga en cuenta que solo se ha utilizado el qbit q0.

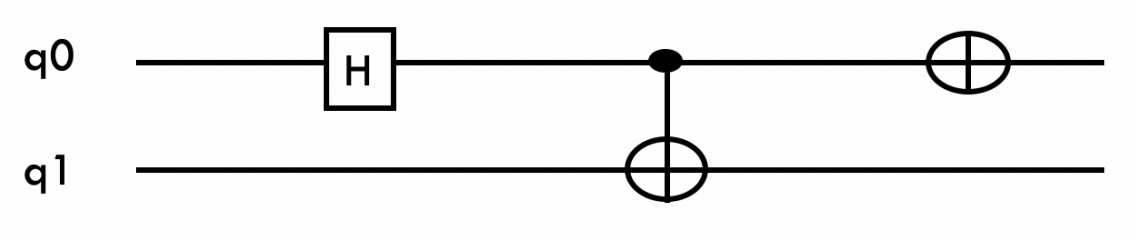
Compruebe el funcionamiento de diferentes puertas cuánticas de 1 y 2 qubits.

6. Generación de números aleatorios con un Computador Cuántico.

Utilizando las puertas Hadamard que sean necesarias, implemente un generador de números aleatorios de 8bit.
Analice los resultados y represente su comportamiento para comprobar si los números son realmente aleatorios.

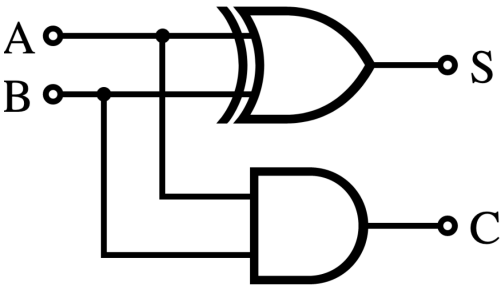
7. Entrelazamiento

Compruebe el comportamiento del siguiente circuito y explique porque el resultado es una función de onda entrelazada.



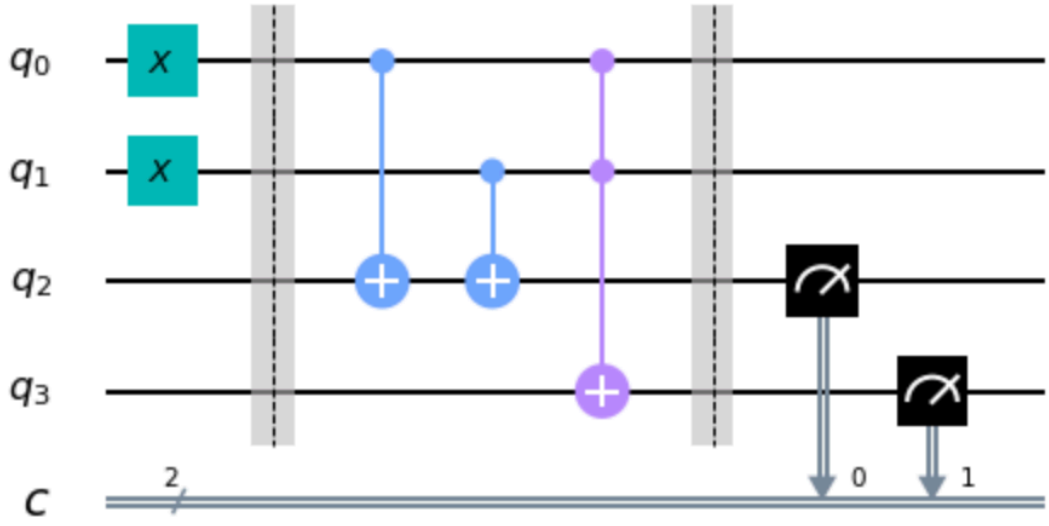
8. Sumador de 2 qbits

Realice el circuito cuántico que equivale a un sumador de dos bits.



Tenga en cuenta que las puertas XOR y AND clásicas no reversibles y por tanto es necesario buscar puertas cuánticas equivalentes pero que sean reversibles.

Puede seguir las indicaciones de la sección
https://qiskit.org/textbook/ch-states/atoms-computation.html



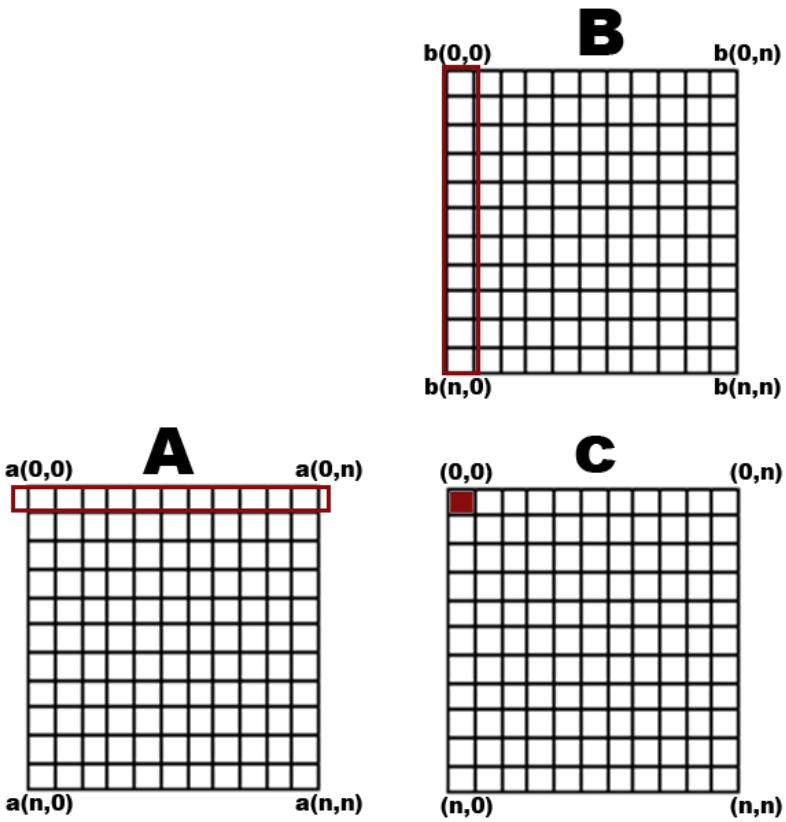
Compruebe su funcionamiento.

Ejercicios opcionales de la práctica 2

Se valorarán hasta **4 puntos** en la nota de la Práctica 2.

9. Producto de matrices en GPU

La multiplicación de matrices es un buen ejemplo de código que se puede paralelizar.



El código C de partida es el siguiente:

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

En la versión inicial del kernel para el producto de matrices podemos considerar multiplicaciones de matrices de NxN y utilizar hilos, de tal manera que el (i,j)-ésimo hilo lee la fila i de la matriz A y la columna j de la matriz B para calcular el dato C[i][j] de la matriz resultante.

Un ejemplo de como realizarlo en CUDA es:

```
#include <stdio.h>
#define N 16

void matrixMultCPU(int a[N][N], int b[N][N], int c[N][N]) {
 int n,m;
 for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
   int sum = 0;
   for (int k = 0; k < N; k++) {
    m = a[i][k];
    n = b[k][j];
    sum += m * n;
   }
   c[i][j] = sum;
  }
 }
}


__global__ void matrixMultGPU(int *a, int *b, int *c) {
 int k, sum = 0;
 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int fil = threadIdx.y + blockDim.y * blockIdx.y;

 if (col < N && fil < N) {
  for (k = 0; k < N; k++) {
   sum += a[fil * N + k] * b[k * N + col];
  }
  c[fil * N + col] = sum;
 }
}


int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;
 int cont,i,j;

 /* inicializando variables con datos*/
 for (i = 0; i < N; i++) {
  cont = 0;
  for (j = 0; j < N; j++) {
   a[i][j] = cont;
   b[i][j] = cont;
   cont++;
  }
 }
```

```
 int size = N * N * sizeof(int);

 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMultGPU<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 // imprimiendo
 for (int y = 0; y < N; y++) {
  for (int x = 0; x < N; x++) {
   printf("[%d][%d]=%d ", y, x, c[y][x]);
  }
  printf("\n");
 }

 return 0;

}
```

En este código, solo se utiliza un bloque con 16x16 Threads y el reparto de trabajo a los threads se ha definido de manera bidimensional.

En la ejecución serie de la CPU las operaciones relizadas son $O(N^3)$ mientras que para la GPU se ha reducido a $O(N^2)$ las operaciones que realiza cada thread.

El trabajo consiste en adaptar el código para analizar el rendimiento obtenido en una GPU según se apliquen diferentes optimizaciones:

- Variar el número de hilos por bloque y ajustar el tamaño de particionado (*tiling*)
- Variar la ubicación de los datos en la jerarquía de memoria de la GPU para optimizar el ancho de banda.
- Optimizaciones adicionales: Aplicar desenrollamiento de bucles (*unrolling*).

Métricas de Rendimiento: Medida de tiempo con CudaEven y con gettimeofday()

```
// Allocate CUDA events that we'll use for timing
    cudaEvent_t start;
    cudaEventCreate(&start);

    cudaEvent_t stop;
    cudaEventCreate(&stop));

    // Record the start event
    cudaEventRecord(start, NULL);

    // Repita la ejecucion del kernel 1000 veces para eliminar
    // efectos de arranque en frio
    int nIter = 1000;
    for (int j = 0; j < nIter; j++)
            kernel <<< grid, threads >>>

    // Record the stop event
    cudaEventRecord(stop, NULL);

    // Wait for the stop event to complete
    cudaEventSynchronize(stop);

    float msecTotal = 0.0f;
    cudaEventElapsedTime(&msecTotal, start, stop);


// Compute and print the performance
    float msecPerKernelExecution = msecTotal / nIter;
    double flopsPerMMull = 2.0 * N * N * N;
    double gigaFlops = (flopsPerMMul * 1.0e-9f) /
                       (msecPerKernelExecution / 1000.0f);
```

Métricas de Rendimiento: FLOPs /MFLOPs/GFLOPs

- **Operaciones en coma Flotante** realizadas (2 x N^3 para multiplicación de matrices NxN) dividido entre el tiempo de ejecución en segundos.
- **MFLOPs = 10^6** FLOPs (millones de FLOPs)
- **GFLOPs =** 10^9 FLOPs

Ejercicios:

Complete los datos relativos a tiempo de ejecución y transferencia de datos en una tabla como la que sigue:

Sin usar memoria compartida	Tiempo de Ejecución (msec)				
Tamaño de la matriz	CPU->GPU	GPU->CPU	Ejecución kernel)	Ratio comparado con 128x128	GFLOPs
16x16					
32x32					
64x64					
128x128				1	
512x512					

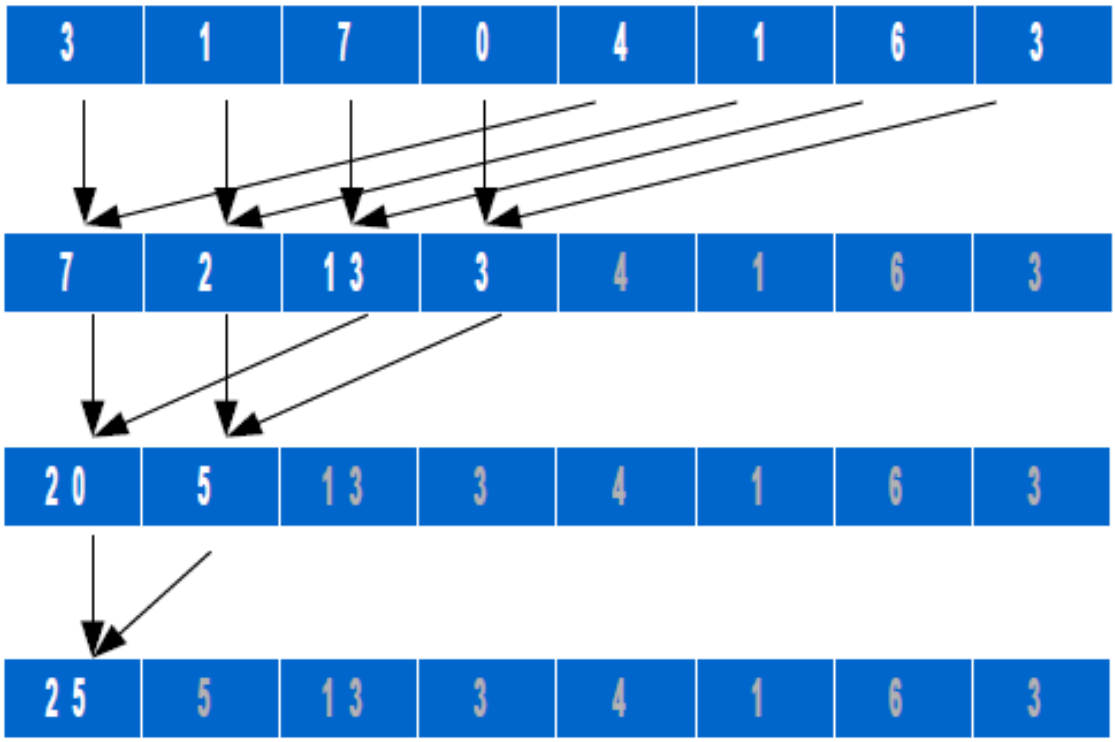
¿Qué conclusiones puedes sacar a partir de estos números?

Con memoria compartida	Tiempo de Ejecución (msec)				
Tamaño de la matriz	CPU->GPU	GPU->CPU	Ejecución kernel)	Ratio comparado con 128x128	GFLOPs
16x16					
32x32					
64x64					
128x128				1	
512x512					

Como ejercicio adicional, prueba a migrar los códigos a aritmética en doble precisión (cambiar "float" por "double" en el tipo de dato de las matrices). ¿En qué medida se ven afectadas las prestaciones?

10. Reducción en GPU

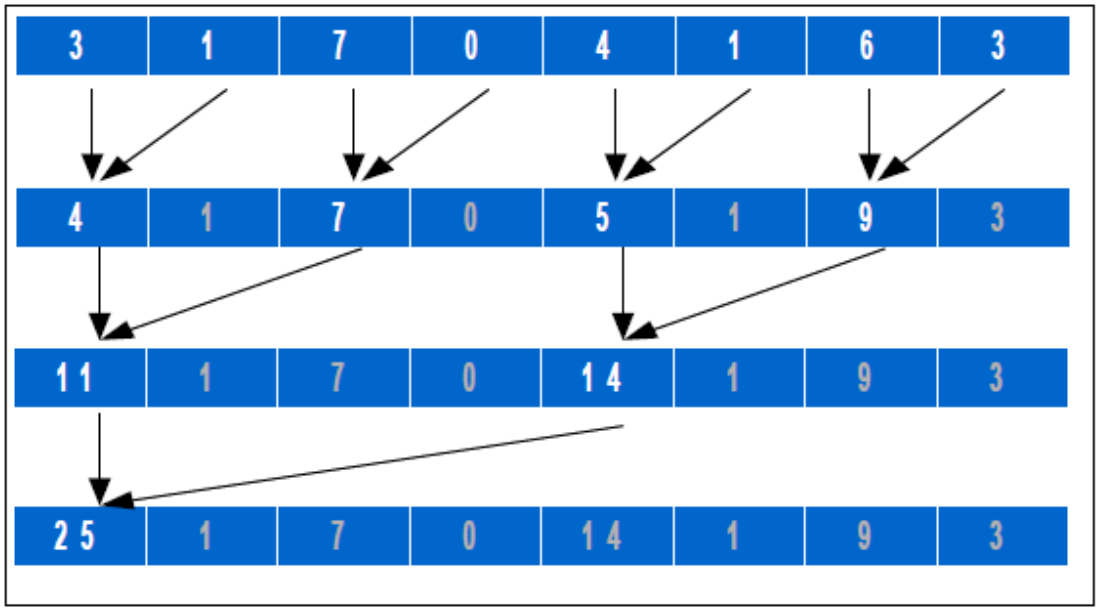
El objetivo de este ejercicio es familiarizarse con un tipo de operaciones muy común en computación científica: **las reducciones**. Una reducción es una combinación de todos los elementos de un vector en un valor único, utilizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular operaciones en paralelo, calculando el resultado en O(logN) pasos sin incrementar el número de operaciones realizadas. Un ejemplo de este tipo de operación se muestra en la siguiente figura:



En este ejercicio se trata de comparar diferentes patrones de acceso a los datos para ir asociando por pares los operandos de cada operación. Esto afecta al rendimiento de la memoria, y también a la complejidad de programación, ya que las expresiones que hay que crear para que los hilos generen los índices de acceso a sus datos en cada paso difieren en dificultad.

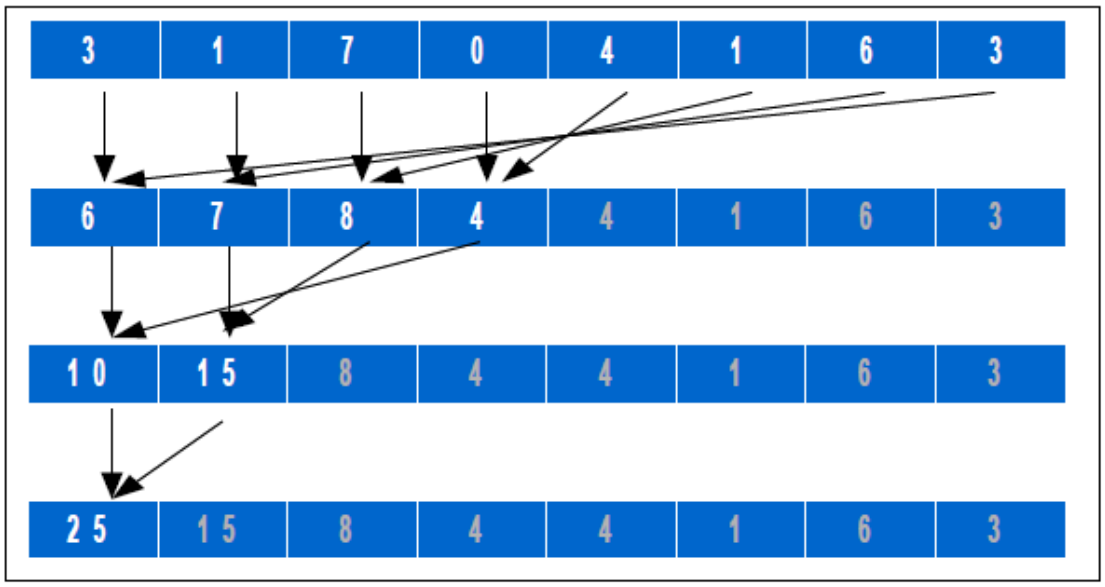
Una vez hayas implementado el esquema de reducción mostrado en la anterior figura, compararemos los resultados con otros esquemas de reducción.

ESQUEMA 2



En el esquema mostrado en la segunda estrategia de reducción, cada hilo es responsable de la suma de dos nodos adyacentes en una distancia potencia de 2 para cada nivel de la operación.

ESQUEMA 3



En la tercera estrategia, cada hilo es responsable de la suma de dos nodos en distancias que van desde n-1 hasta 1. Por ejemplo, en el primer nivel

de la operación, el hilo número 1 suma los elementos 1 y 8, siendo la distancia entre los dos nodos 8-1 = 7.
El hilo número 2 suma los elementos 2 y (8-1). La distancia correspondiente es pues 7-2 = 5. Y las distancias para los nodos sumados por los hilos número 3 y 4 son 6-3 = 2 y 5-4 = 1, respectivamente.

El patrón de acceso regular en ambos esquemas. La mayoría de aplicaciones utilizan patrones de este tipo para eliminar divergencias innecesarias o conflictos en accesos a bancos de memoria.

Se proporciona un ejemplo que ya implementa para N=512 elementos diferentes esquemas de reducción. Se Pide

- Entender el comportamiento de cada uno de ellos, explicando las diferencias que encuentra.
- Realizar un análisis del efecto en el rendimiento por los accesos a memoria y la divergencia de Threads.
- Generalizar el funcionamiento del más eficiente para cualquier tamaño de elementos, N >> Tamaño de bloque

11. Revisión de los ejemplos instalados con CUDA

Consulte los ejemplos disponibles en su instalación de CUDA.

```
%cd /usr/local/cuda/samples/
```

```
[ ] %cd /usr/local/cuda/samples/

    /usr/local/cuda-10.1/samples

    !ls -la /usr/local/cuda/samples/0_Simple/matrixMul/

    total 40
    drwxr-xr-x  2 root root  4096 Oct 14 16:25 .
    drwxr-xr-x 52 root root  4096 Oct 14 16:25 ..
    -rw-r--r--  1 root root 10945 Aug  9  2019 Makefile
    -rw-r--r--  1 root root 10962 Aug  9  2019 matrixMul.cu
    -rw-r--r--  1 root root  2320 Aug  9  2019 NsightEclipse.xml
    -rw-r--r--  1 root root   434 Aug  9  2019 readme.txt

[ ] !cat /usr/local/cuda/samples/0_Simple/matrixMul/readme.txt

    Sample: matrixMul
    Minimum spec: SM 3.0

    This sample implements matrix multiplication which makes use of shared memory to ensure data reuse, the matrix multiplication

    Key concepts:
    CUDA Runtime API
    Linear Algebra

[ ] !cat /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul.cu

    /**
     * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
     *
     * Please refer to the NVIDIA end user license agreement (EULA) associated
     * with this source code for terms and conditions that govern your use of
```

```
[ ] %cd /content/
```

```
/content
```

```
%%writefile matmul.cu
/**
 * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Matrix multiplication: C = A * B.
 * Host code.
 *
 * This sample implements matrix multiplication which makes use of shared memory
 * to ensure data reuse, the matrix multiplication is done using tiling approach.
 * It has been written for clarity of exposition to illustrate various CUDA progr
 * principles, not with the goal of providing the most performant generic kernel
 * See also:
 * V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra,"
 * in Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC '08),
 * Piscataway, NJ: IEEE Press, 2008, pp. Art. 31:1-11.
 */

// System includes
#include <stdio.h>
#include <assert.h>
```

Compilar con

```
!/usr/local/cuda/bin/nvcc -I /usr/local/cuda/samples/common/inc -
arch=sm_35 -rdc=true matmul.cu -o matmul -lcudadevrt
```

Probar su funcionamiento y hacer perfilado

```
[ ] !./matmul

    [Matrix Multiply Using CUDA] - Starting...
    GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

    MatrixA(320,320), MatrixB(640,320)
    Computing result using CUDA Kernel...
    done
    Performance= 1410.34 GFlop/s, Time= 0.093 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
    Checking computed result for correctness: Result = PASS

    NOTE: The CUDA Samples are not meant for performancemeasurements. Results may vary when GPU Boost is e
```

```
!nvprof ./matmul

[Matrix Multiply Using CUDA] - Starting...
==834== NVPROF is profiling process 834, command: ./matmul
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 1396.75 GFlop/s, Time= 0.094 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performancemeasurements. Results may vary when GPU Boost is e
==834== Profiling application: ./matmul
==834== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.39%  28.013ms       301  93.067us  90.881us  94.241us  void MatrixMulCUDA<int=32
                    0.39%  108.74us         2  54.368us  36.544us  72.192us  [CUDA memcpy HtoD]
                    0.22%  63.136us         1  63.136us  63.136us  63.136us  [CUDA memcpy DtoH]
      API calls:   85.74%  182.27ms         3  60.757ms  3.7910us  182.26ms  cudaMalloc
                   12.34%  26.241ms         1  26.241ms  26.241ms  26.241ms  cudaEventSynchronize
                    0.89%  1.9024ms       301  6.3200us  5.0310us  100.19us  cudaLaunchKernel
                    0.47%  990.15us         3  330.05us  105.69us  651.04us  cudaMemcpy
                    0.22%  463.37us         1  463.37us  463.37us  463.37us  cuDeviceTotalMem
                    0.12%  244.59us         2  122.29us  108.57us  136.02us  cudaGetDeviceProperties
                    0.09%  198.41us         3  66.137us  8.1960us  159.80us  cudaFree
                    0.07%  143.20us        97  1.4760us     154ns  57.449us  cuDeviceGetAttribute
                    0.03%  73.288us         1  73.288us  73.288us  73.288us  cudaDeviceSynchronize
                    0.01%  15.016us         1  15.016us  15.016us  15.016us  cuDeviceGetName
```

También es posible compilar con el makefile ya existente para compilar

```
%cd /usr/local/cuda/samples/0_Simple/matrixMul
!ls
```

```
/usr/local/cuda-10.1/samples/0_Simple/matrixMul
Makefile  matrixMul  matrixMul.cu  matrixMul.o  NsightEclipse.xml  readme.txt
```

```
!cat Makefile
```

```
[3] !make
    !ls -la
```

```
/usr/local/cuda-10.1/bin/nvcc -ccbin g++ -I../../common/inc  -m64    -gencode arch=compute_30,code=sm_30 -genc
/usr/local/cuda-10.1/bin/nvcc -ccbin g++   -m64      -gencode arch=compute_30,code=sm_30 -gencode arch=compute
mkdir -p ../../bin/x86_64/linux/release
cp matrixMul ../../bin/x86_64/linux/release
total 840
drwxr-xr-x 1 root root   4096 Oct 23 12:25 .
drwxr-xr-x 1 root root   4096 Oct 14 16:25 ..
-rw-r--r-- 1 root root  10945 Aug  9  2019 Makefile
-rwxr-xr-x 1 root root 721760 Oct 23 12:25 matrixMul
-rw-r--r-- 1 root root  10962 Aug  9  2019 matrixMul.cu
-rw-r--r-- 1 root root  87856 Oct 23 12:25 matrixMul.o
-rw-r--r-- 1 root root   2320 Aug  9  2019 NsightEclipse.xml
-rw-r--r-- 1 root root    434 Aug  9  2019 readme.txt
```

```
[4] !./matrixMul
```

```
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 1663.07 GFlop/s, Time= 0.079 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

Si desea compilar un programa que no está en los ejemplos puede hacerlo siguiendo los siguiente pasos:

1,-coloquese en un directorio de trabajo (por ejemplo /content/workcuda) Escriba el fichero con

%%writefile

```
%%writefile labsolReduction0.cu

// includes, kernels
#include <stdio.h>
#include <assert.h>

#define NUM_ELEMENTS 512


// **===-----------------------------------------------------------===**
//! @param g_idata  input data in global memory
//                  result is expected in index 0 of g_idata
//! @param n        input number of elements to scan from input data
// **===-----------------------------------------------------------===**

__global__ void reduction(float *g_data, int n)
{
  int stride;
```

Compile con los Include necesarios (Opción -I)

```
!/usr/local/cuda/bin/nvcc -I /usr/local/cuda/samples/common/inc
-arch=sm_35 -rdc=true labsolReduction0.cu -o labsolReduction0
-lcudadevrt
```

Ejecute el programa y realice análisis de rendimiento con nvprof

```
!./labsolReduction0
```

```
Test PASSED
device: 130816.000000  host: 130816.000000
```

```
[ ] !nvprof ./labsolReduction0
```

```
==2810== NVPROF is profiling process 2810, command: ./labsolReduction0
Test PASSED
device: 130816.000000  host: 130816.000000
==2810== Profiling application: ./labsolReduction0
==2810== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   68.19%  8.0960us         1  8.0960us  8.0960us  8.0960us  reduction(float*, int)
                   16.17%  1.9200us         1  1.9200us  1.9200us  1.9200us  [CUDA memcpy HtoD]
                   15.63%  1.8560us         1  1.8560us  1.8560us  1.8560us  [CUDA memcpy DtoH]
      API calls:   99.47%  188.56ms         1  188.56ms  188.56ms  188.56ms  cudaMalloc
                    0.30%  572.94us         1  572.94us  572.94us  572.94us  cuDeviceTotalMem
                    0.13%  246.88us        97  2.5450us     177ns  107.95us  cuDeviceGetAttribute
```

12. Otros Ejemplos de GPGPU

- Programming Massively Parallel Processors : A Hands-on Approach David B. Kirk , Wen-Mei W. Hwu

  https://www.iaa.csic.es/~dani/ebooks/MK.Programming.Massively.Parallel.Processors.2nd.Edition.Dec.2012.pdf

- https://docs.nvidia.com/cuda/cuda-samples/index.html

- http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf

13. Algoritmos Cuánticos

Se pide entender e implementar el comportamiento de los siguientes algoritmos cuánticos.

2. Algoritmo de Bernstein-Vazirani.
3. Algoritmo de Phase Kick-Back
4. Teleportación Cuántica.
5. Algoritmo de Grover: búsqueda general
6. Algorimtmo QFT: Transformada de Fourier Cuántica
7. Algoritmo de Shor : factorización de números
8. Algoritmo QPE: Cálculo del número PI

14. Otros Ejemplos de Computación Cuántica

Tutoriales de computación cuántica en QisKit

- Computación Cuántica en Finanzas
- Machine Learning
- Ruido
- Química
- Simuladores

Referencias:

- https://qiskit.org/textbook/preface.html
- Quantum Computation and Quantum Information by Michael A. Nielsen & Isaac L. Chuang