Programación GPGPU y Computación Cuántica
Procesamiento de Datos a Gran Escala

José Antonio Álvarez Ocete
Francisco Javier Sáez Maldonado

7 de noviembre de 2021

Índice

Introducción

Esta práctica está dividida en tres partes:

1. La primera se concentrará en realizar diversas pruebas con los elementos más básicos de la programación en CUDA.
2. La segunda parte está centrada en el estudio de la computación cuántica a través de Qiskit y el Quantum Composer de IBM.
3. La tercera queda dedicada al estudio de dos de los más famosos algoritmos cuánticos: el algoritmo de teleportación cuántico y el algoritmo de Grover.

1. Parte 1: Programación GPGPU

1.1. Recursos de la GPU

Ejercicio 1

Obtener información sobre los recursos de la GPU

Para programar en GPU, el primer paso que debemos dar es conocer las características (y por tanto, los recursos) que el dispositivo del que disponemos nos ofrece. En este caso práctico, vamos a realizar las pruebas utilizando Google Colaboratory. Google pone a disposición del usuario tanto GPUs como TPUs que son más que suficientes para realizar algunas pruebas.

Lo primero que haremos es estudiar qué recursos tiene el equipo que nos ofrece Google. la GPU que nos ofrecen desde Google. Podemos ejecutar el comando `!lscpu` para obtener información sobre el modelo de CPU que tenemos. El resultado nos muestra que tenemos un procesador *Intel(R) Xeon(R) CPU @ 2.30GHz* de 64 bits. Además, si ejecutamos `!free -kh` vemos que tenemos los siguientes recursos disponibles:

```
              total        used        free      shared  buff/cache   available
Mem:            12G        571M        9.9G        1.2M        2.2G         11G
Swap:            0B          0B          0B
```

Como vemos, tenemos 12Gb de memoria prácticamente disponibles. Ahora, queremos comprobar también los recursos GPU que tenemos disponibles, sabiendo que hemos activado el entorno de Colaboratory para que tenga GPU. Verificamos las tarjetas gráficas que tenemos utilizando la orden `nvidia-smi`. Obtenemos el resultado siguiente:

```
       Fri Oct 29 14:57:32 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.29.05    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   48C    P8    30W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Como vemos, en **esta conexión al servidor** se nos ha proporcionado una *Nvidia Tesla K80*. Hacemos hincapié en que es en esta conexión porque en Google Colaboratory se asigna una GPU que esté disponible en ese momento, por lo que en otra ejecución podríamos obtener otra de capacidades de computación similares.
Además de eso, se nos indica que tenemos como driver instalado la versión `11.2` de CUDA. Sin embargo, hemos comprobado que la versión de ejecución de CUDA no es la misma de dos maneras distintas. Primeramente, si hacemos !nvcc −−version ya se nos indica por primera vez lo siguiente:

```
Cuda compilation tools, release 11.1, V11.1.105
Build cuda_11.1.TC455_06.29190527_0
```

Además de eso, podemos compilar el ejemplo que tenemos en /usr/local/cuda/samples/1_Utilities/deviceQuery/ usando el `makefile` proporcionado y, al ejecutar el ejemplo, obtenemos la siguiente salida (mostramos un resumen de la misma debido a su extensión):

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.2 / 11.1
  ...
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        114688 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ...
```

Como datos a destacar, vemos que tenemos 65536 registros disponibles por bloque, con un tamaño de Warp de 32 y con un máximo número de hilos por bloque de 1024. Estos datos tendrán implicaciones a la hora de hacer ejecuciones. Además, viendo la dimensión máxima de un grid en la tripla que se nos ofrece, obtenemos que el número máximo de bloque es 65535.

1.2. Suma de 2 vectores

Ejercicio 2

Escribe un programa en CUDA que realice la suma de dos vectores

1.2.1. Suma de dos enteros

Comenzamos con un ejemplo sencillo de cálculo de una suma de enteros usando CUDA. El código está proporcionado en el cuaderno de enunciado de la práctica. Explicamos brevemente el código para entender mejor las preguntas:

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

Este bloque implementa la función suma de dos enteros que se pasan como un puntero. Se almacena el valor de la suma en un tercer puntero que se pasa a la función.

```
// host copies of variables a, b & c
int a, b, c;

// device copies of variables a, b & c
int *d_a, *d_b, *d_c;

int size = sizeof(int);
// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Setup input values
c = 0;
a = 3;
b = 5;
```

En este bloque se hace la declaración de variables. Se hacen tres variables, luego se declaran punteros a tres variables y se declara un entero `size` que tiene como valor el tamaño de un entero. A continuación, se declaran en CUDA punteros (usando los tres punteros declarados anteriormente) de tamaño `size`. Por último, se le da valor a los enteros para poder sumarlos.

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);
```

En este fragmento, primero se copia a la memoria de la GPU las referencias a los enteros que hemos inicializado anteriormente, es decir, ahora en la memoria de CUDA los punteros inicializados en cuda apuntan a la misma posición de memoria que tiene en el disco los valores que se quieren sumar. Se usa el valor `cudaMemcpyHostToDevice` del enumerado `cudaMemcpyKind` que indica que el valor irá desde el Host (el equipo) al Device (la GPU). A continuación, se realiza la operación suma `add<<<1,1>>>` indicándole que se haga con 1 bloque y 1 hebra.

```
// Copy result back to host
cudaError err = cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess)
        printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));

printf("result is %d\n",c);
// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

En este último código, se hace apuntar el puntero que tiene el resultado en la GPU al puntero que tiene la memoria del dispositivo, guardando si ha habido algún error. Ahora, se usa otro valor del enumerado: `cudaMemcpyDeviceToHost`, que indica que el valor vaya desde la GPU al equipo. Finalmente, se liberan de la GPU los espacios reservados para los punteros. El resultado de la ejecución es el esperado.

```
!./suma1d

result is 8
```

Si comprobamos el perfil de ejecución, vemos el siguiente resultado:

```
        ==229== NVPROF is profiling process 229, command: ./suma1d
==229== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
result is 8
==229== Profiling application: ./suma1d
==229== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   37.50%  3.7440us         2  1.8720us  1.5040us  2.2400us  [CUDA memcpy HtoD]
                   36.54%  3.6480us         1  3.6480us  3.6480us  3.6480us  add(int*, int*, int*)
                   25.96%  2.5920us         1  2.5920us  2.5920us  2.5920us  [CUDA memcpy DtoH]
      API calls:   99.56%  270.53ms         3  90.177ms  2.6620us  270.52ms  cudaMalloc
                    0.21%  565.39us         1  565.39us  565.39us  565.39us  cuDeviceTotalMem
                    0.12%  321.01us       101  3.1780us     210ns  140.66us  cuDeviceGetAttribute
                    0.06%  171.37us         3  57.124us  5.6250us  152.77us  cudaFree
                    0.02%  60.590us         3  20.196us  11.684us  25.788us  cudaMemcpy
```

```
 0.01%  32.193us         1  32.193us  32.193us  32.193us  cuDeviceGetName
 0.01%  24.277us         1  24.277us  24.277us  24.277us  cudaLaunchKernel
 0.00%  5.0420us         1  5.0420us  5.0420us  5.0420us  cuDeviceGetPCIBusId
 0.00%  1.9970us         3     665ns     186ns  1.0810us  cuDeviceGetCount
 0.00%  1.5280us         2     764ns     401ns  1.1270us  cuDeviceGet
 0.00%     420ns         1     420ns     420ns     420ns  cuDeviceGetUuid
```

Como vemos, tenemos una única llamada a la función `add`, 3 llamadas a la función `cudaMalloc` y otras tres para copiar las referencias usando `cudaMemcpy`. Podemos observar en esta primera ejecución cómo se hacen 101 llamadas a `cuDeviceGetAttribute`, que nos devuelve un valor que se almacena en la memoria de la GPU.

1.2.2. Paralelizando

Sumando vectores

El objetivo ahora será modificar el código actual para que la suma de vectores se realice según los bloques y las hebras que se le indiquen. En particular, querremos modificar el número de hebras para que se puedan paralelizar las operaciones. Comenzamos cambiando el número de bloques a 512. Podemos apreciar en el código de la suma un cambio:

```
__global__ void add(int *a, int *b, int *c) {
    //*c = *a + *b;
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Ahora, accedemos al valor `blockIdx.x` de cada uno de los punteros para realizar la suma. Esto es importante pues ahora querremos que cada operación la realice un bloque. Generalizamos el programa anterior para que sume vectores del mismo tamaño, elemento a elemento.

```
int *a, *b, *c;         // host copies of variables a, b & c
int *d_a, *d_b, *d_c;   // device copies of variables a, b & c
int size = N * sizeof(int);
// Allocate space for host copies of a, b, c   Setup inputvalues
a =  (int *) malloc(size);
b =  (int *) malloc(size);
c =  (int *) malloc(size);
//   Setup input values
for( int i = 0; i < N; i++ ){
        a[i] = i;
          b[i] = N-i;
        c[i] = 0;
}
```

Como vemos, la inicialización de los valores es diferente. Ahora necesitamos dejar espacio en el disco que tenga tamaño: el tamaño del vector por lo que ocupa un entero. Entonces, la inicialización se hace en un bucle `for`.

```
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
```

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

Se reservan en memoria espacio para los vectores y se copian a la GPU usando la misma función, solo que asignando más espacio mediante el parámetro `size`.

```
// Launch add() kernel on GPU  Se lanzan N bloques de 1 Thread.
add<<<N,1>>>(d_a, d_b, d_c);
```

Se realiza la operación suma entre los dos vectores. Llamándola de la forma `add<<<N,M>>>` estamos indicando que queremos realizar la operación usando N bloques y M hebras. En este caso concreto, estaríamos diciendo que se haga la operación con $N = 512$ (definido como variable del programa) bloques y 1 hebra.
El resultado final del programa es el siguiente:

```
valor a[0] es 0
valor b[0] es 512
resultado c[0] es 512
valor a[2] es 2
valor b[2] es 510
resultado c[2] es 512
```

Como podemos ver, todo parece estar ocurriendo en orden. Añadimos un bucle for que comprueba si todos los valores del vector `c` valen lo mismo, y lo imprime por pantalla:

```
bool res = 1;
for(int i  = 0; i < N; i++)
  if(c[i] != N)
        res = 0;

printf("Todos los valores de c valen lo mismo (1 true, 0 false):%d\n",res);
...
Todos los valores de c valen lo mismo (1 true, 0 false): 1
```

Como vemos, el resultado es correcto. Vemos el perfil de ejecución en este caso:

```
==1891== NVPROF is profiling process 1891, command: ./suma2dvector
==1891== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
 valor a[0] es 0
 valor b[0] es 512
resultado c[0] es 512
 valor a[2] es 2
 valor b[2] es 510
resultado c[2] es 512
Todos los valores de c valen lo mismo (1 true, 0 false): 1
==1891== Profiling application: ./suma2dvector
==1891== Profiling result:
        Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU:    40.98%  5.0880us         1  5.0880us  5.0880us  5.0880us  add(int*, int*, int*)
        37.63%  4.6720us         2  2.3360us  2.0160us  2.6560us  [CUDA memcpy HtoD]
```

```
        21.39%  2.6560us         1  2.6560us  2.6560us  2.6560us  [CUDA memcpy DtoH]
API calls:  99.49%  190.44ms         3  63.479ms  2.2440us  190.43ms  cudaMalloc
                0.28%  529.05us         1  529.05us  529.05us  529.05us  cuDeviceTotalMem
                0.10%  199.48us       101  1.9750us     145ns  87.079us  cuDeviceGetAttribute
                0.07%  134.29us         3  44.764us  3.2210us  115.10us  cudaFree
                0.03%  56.462us         3  18.820us  11.545us  25.237us  cudaMemcpy
                0.01%  26.486us         1  26.486us  26.486us  26.486us  cuDeviceGetName
                0.01%  25.925us         1  25.925us  25.925us  25.925us  cudaLaunchKernel
                0.00%  5.9010us         1  5.9010us  5.9010us  5.9010us  cuDeviceGetPCIBusId
                0.00%  1.7020us         3     567ns     162ns     876ns  cuDeviceGetCount
                0.00%  1.2500us         2     625ns     233ns  1.0170us  cuDeviceGet
                0.00%     385ns         1     385ns     385ns     385ns  cuDeviceGetUuid
```

Como vemos, de nuevo tenemos una única llamada a la función `add`. No se observa una diferencia apreciable entre el tiempo de ejecución de esta llamada en este caso y en el anterior. Tampoco en el resto de operaciones que se hacen en la GPU.

1 Bloque - N Threads

El objetivo será usar la modificación del programa que utiliza en un solo bloque N hebras para paralelizar la ejecución. Habíamos definido antes $N = 512$ para usar ese número de bloques. Ahora, se redefine como $N = 1024$ para usar este número de hebras (que vimos en la descripción de la GPU que es el número máximo).
Al igual que en el caso anterior, la función suma cambia:

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Como vemos, ahora se accede al índice de la hebra usando `threadIdx.x`. Otro cambio que se debe hacer claramente es que la función add se llame de la forma:

```
// Launch add() kernel on GPU  Se lanzan 1 bloques de N Threads.
add<<<1,N>>>(d_a, d_b, d_c);
```

Volvemos a mostrar el resultado de la ejecución de este nuevo programa.

```
valor a[10] es 10
valor b[10] es 1014
resultado c[10] es 1024
valor a[0] es 0
valor b[0] es 1024
resultado c[0] es 1024
Todos los valores de c valen lo mismo (1 true, 0 false): 1
```

Se aprecia que todos los resultados son correctos. Se pide de nuevo que se compare el perfil de ejecución con los anteriores, pero no se aprecia ninguna diferencia significativa en los resultados.

1.2.3. Preguntas

1. Pruebe a lanzar diferente número de Threads (con un solo 1 bloque) ¿Cual son los valores máximos y mínimos de número de theads por bloque en esta GPU?

Estos valores mínimos ya se podían observar cuando ejecutábamos el programa `deviceQuery`. En él, aparecía que el número máximo de threads por bloque era 1024. Ya hemos mostrado que, en el caso de ejecutarlo con este número los resultados son correctos.

```
add<<<1,1025>>>(d_a, d_b, d_c);
...
valor a[10] es 10
 valor b[10] es 1014
resultado c[10] es 0
 valor a[0] es 0
 valor b[0] es 1024
resultado c[0] es 0
Todos los valores de c valen lo mismo (1 true, 0 false): 0
```

Como vemos, en este caso no se calculan bien los valores aunque estén bien inicializados y por tanto la operación falla. De hecho, hemos comprobado que para todo número diferente de 1024, esta operación falla. Esto además tiene sentido pues recordamos que en la suma estamos accediendo al índice de la hebra, por lo que no tendremos suficientes índices de hebras para acceder a todos los elementos del vector. Además, hemos comprobado cuántos elementos se suman correctamente cambiando el bucle de comprobación de la siguiente forma:

```
bool res = 1;
int well = 0;
  for(int i  = 0; i < N; i++)
    if(c[i] == N){
      well +=1;
    }
res = well == N;
printf("Todos los valores de c valen lo mismo (1 true, 0 false): %d\n",res);
printf("Numero de elementos sumados correctamente: %d\n",well);
```

Se comprueba que, para cualquier $n < N$, el número de elementos correctamente sumados es justamente n. Por ejemplo, en el caso $n = 3$:

```
add<<<1,3>>>(d_a, d_b, d_c);
...
   Todos los valores de c valen lo mismo (1 true, 0 false): 0
   Numero de elementos sumados correctamente: 3
```

2. Pruebe a lanzar diferente número de bloques (con un solo thread) ¿Cual son los valores máximos y mínimos de número de bloques en esta GPU?

   El número máximo de bloques ya lo habíamos visto en la información inicial. Desde las capacidades computacionales 3,0, la dimensión x de un grid de bloques de hebras puede ser 2^{31-1}.

1.3. $NUMBLOCK$ bloques y $NUMTHREADS$ hilos

Pretendemos ahora lanzar $NUMBLOCK \in \mathbb{N}$ bloques por $NUMTHREAD \in \mathbb{N}$ hilos. Lo primero que debemos ahcer es cambiar la función suma para que se pueda acceder en cada hebra al elemento correspondiente. Recordamos que una matriz $N \times M$ se puede representar como un vector de NM posiciones. Si quisiésemos acceder a la posición z de un vector utilizando bloques y hebras, debemos descomponer ese z como

$$z = n * i + j$$

donde n será el tamaño de bloque, i el número de bloque en el que está el elemento y j la hebra que tendrá que manejarlo. Podemos realizar esto en la nueva funcion add que redefinimos, usando como $n = blockDim.x$, $i = blockIdx.x$ y $j = threadID.x$:

```
__global__ void add(int *a, int *b, int *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
         c[index] = a[index] + b[index];
}
```

Ahora, debemos también definir los nuevos tamaños de bloque y de número de hebras. Debemos adecuar este tamaño al tamaño de nuestro vector. Llamemos N al tamaño del vector (o matriz vectorizada) que queremos sumar. Sea NT el número de hebras. Supongamos que queremos dividir este trabajo para que cada hebra realice (si es posible) una única operación. Se podría pensar a priori que el número de bloques que necesitamos para realizar nuestra operación es:

$$NB = \frac{N}{NT}.$$

Sin embargo, Esta división podría no ser entera al no ser N múltiplo de NT. En esos casos, no podríamos acceder a todos los elementos de nuestro vector y no tendríamos por tanto un resultado correcto. Es por ello que debemos considerar entonces como número óptimo de bloques:

$$NB = \frac{N + NT - 1}{NT} = \frac{N}{NT} + 1 - \frac{1}{NT}.$$

Si aplicamos la función parte entera a este número, obtendríamos K como valor. En el código proporcionado, tenemos el tamaño del vector fijo $N = 1024^2$ y el número de hebras por bloque $NT = 512$, resultando en una división exacta $NB = 2048$. Así, se puede llamar directamente a la función `add` del siguiente modo:

```
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

Ejercicio 3

Qué ocurre cuando se aumenta N?

Con el código actual si aumentamos N, el cociente no será exacto y tendremos que habrá datos que queden sin calcular. Por ejemplo, si ponemos $N = 1025^2$ obtenemos la siguiente salida:

```
Todos los valores de c valen lo mismo (1 true, 0 false): 0
```

Que es un resultado negativo en cuanto a que todos los valores valen lo mismo. Lo mismo ocurre cuando usamos un número menor de N, pues se nos desajusta la exactitud del número de bloques.
Por ello, debemos actualizar el código para que se calcule bien el número de bloques que necesitamos según el número de hebras que hemos establecido, usando la siguiente fórmula que ya hemos explicado anteriormente:

```
add<<<(N + THREADS_PER_BLOCK - 1)/THREADS_PER_BLOCK,
                          THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

Si volvemos a ejecutar el código, esta vez obtenemos un 1 en la salida, indicando que la operación se ha hecho correctamente.

1.4. Suma de 2 matrices

Vamos ahora a hacer una implementación de una suma de matrices. Esto no es más que una extensión del ejercicio anterior, pues sabemos que además las matrices se representan en memoria como vectores de dimensión $R \times C$ donde R es el número de filas y C el de columnas.
Nuestro interés estará en conseguir, usando grids de bloques y hebras, que el la hebra que esté en la posición (i, j) en el grid de bloques-hebras, realice la suma $c_{i,j} = a_{i,j} + b_{i,j}$. Para llevar esto a cabo, usaremos bloques de dimensión 8×8, pues hemos visto en la información inicial que el tamaño máximo de un bloque de hebras es $(x, y, z) = (1024, 1024, 64)$. Así, podremos usar 16 bloques en cada SM sin tener errores.
Para calcular el número de bloques, lo que haremos será usar la fórmula anterior tanto por filas como por columnas, obteniendo:

$$(R + 63)/64 \qquad y(C + 63)/64$$

bloques. Escribimos entonces nuestra función del kérnel como sigue:

```
__global__ void add(int *a, int *b, int *c, int r, int col) {
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;
    int j = (blockIdx.y * blockDim.y) + threadIdx.y;
    int index = col*i + j;

   # Comprobamos que no nos salimos de la matriz
    if (i < r && j < col)
      c[index] = a[index] + b[index];
}
```

Inicializamos la matriz a con todas las posiciones a 2 y la matriz b con todas las posiciones a 3, simplemente por cambiar de valores respecto al ejercicio anterior. Vamos a comenzar sumando matrices cuadradas de 1024×1024, aunque el código debe ser igualmente válido para matrices no cuadradas y/o cuyas dimensiones no sean potencias de 2 (lo cual podríamos pensar que puede dar error al estar divididos los bloques de hebras en estas potencias).
Seguimos también los mismos pasos que hicimos en el ejercicio anterior:

```
int *a, *b, *c;         // host copies of variables a, b & c
int *d_a, *d_b, *d_c;   // device copies of variables a, b & c
int size = R*C*sizeof(int);
// int N = R*C;

// Allocate space for host copies of a, b, c   Setup input values
a =  (int *) malloc(size);
b =  (int *) malloc(size);
c =  (int *) malloc(size);
// Setup input values

for( int i = 0; i < R; i++ ){
    for(int j = 0; j < C; j++){
        a[i*C + j] = 2;
        b[i*C + j] = 3;
    }
}
// Allocate space for device copies of a, b, c
cudaMalloc((void **) &d_a, size);
cudaMalloc((void **) &d_b, size);
cudaMalloc((void **) &d_c, size);

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

En este bloque hemos inicializado la memoria en el host, inicializado las variables y luego las hemos copiado a la memoria de la GPU.

```
dim3 dimGrid((R + THREADS_PER_BLOCK - 1)/THREADS_PER_BLOCK,
                          (C + THREADS_PER_BLOCK -1)/THREADS_PER_BLOCK, 1);

dim3 dimBlock(THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1);

add<<<dimGrid,dimBlock>>>(d_a, d_b, d_c,R,C);

cudaDeviceSynchronize();
```

En este bloque, declaramos las dimensiones del grid y de los bloques y ejecutamos el kernel. Tras la ejecución, el profiler nos muestra lo siguiente:

```
==1114== NVPROF is profiling process 1114, command: ./sumaMatrices
==1114== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
Número de errores: 0
==1114== Profiling application: ./sumaMatrices
==1114== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.13%  1.5977ms         1  1.5977ms  1.5977ms  1.5977ms  [CUDA memcpy DtoH]
                   39.13%  1.1991ms         2  599.57us  586.88us  612.26us  [CUDA memcpy HtoD]
```

```
                    8.74%  267.84us         1  267.84us  267.84us  267.84us  add(int*, int*, int*,
      API calls:   96.00%  194.95ms         3  64.984ms  122.30us  194.70ms  cudaMalloc
                    2.07%  4.2101ms         3  1.4034ms  707.86us  2.7744ms  cudaMemcpy
                    0.86%  1.7499ms         1  1.7499ms  1.7499ms  1.7499ms  cuDeviceTotalMem
                    0.68%  1.3737ms         3  457.91us  164.43us  607.69us  cudaFree
                    0.19%  388.20us         1  388.20us  388.20us  388.20us  cudaDeviceSynchronize
                    0.16%  332.80us       101  3.2950us     235ns  155.81us  cuDeviceGetAttribute
                    0.02%  35.570us         1  35.570us  35.570us  35.570us  cuDeviceGetName
                    0.01%  28.411us         1  28.411us  28.411us  28.411us  cudaLaunchKernel
                    0.00%  6.8700us         1  6.8700us  6.8700us  6.8700us  cuDeviceGetPCIBusId
                    0.00%  2.2460us         3     748ns     198ns  1.0250us  cuDeviceGetCount
                    0.00%  1.5120us         2     756ns     404ns  1.1080us  cuDeviceGet
                    0.00%     515ns         1     515ns     515ns     515ns  cuDeviceGetUuid
```

Como vemos, no tenemos errores en la ejecución. Podemos probar a cambiar las dimensiones poniendo una matriz no cuadrada. Modificamos el archivo directamente usando la orden de bash 'sed'.

```
!sed -i '/#define R/c\#define R 512' sumaMatrices.cu
!sed -i '/#define C/c\#define C 1024' sumaMatrices.cu
!/usr/local/cuda/bin/nvcc -arch=sm_35 -rdc=true sumaMatrices.cu -o sumaMatrices_ch
!nvprof ./sumaMatrices_changedim
```

La salida mostrada por el profiler es la siguiente:

```
==708== NVPROF is profiling process 708, command: ./sumaMatrices_changedim
==708== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
Número de errores: 0
==708== Profiling application: ./sumaMatrices_changedim
==708== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   59.12%  569.54us         2  284.77us  282.66us  286.88us  [CUDA memcpy HtoD]
                   28.03%  270.02us         1  270.02us  270.02us  270.02us  [CUDA memcpy DtoH]
                   12.85%  123.74us         1  123.74us  123.74us  123.74us  add(int*, int*, int*,
      API calls:   98.16%  203.58ms         3  67.859ms  124.40us  203.27ms  cudaMalloc
                    0.96%  1.9813ms         3  660.42us  291.06us  1.2887ms  cudaMemcpy
                    0.37%  767.02us         3  255.67us  137.30us  318.07us  cudaFree
                    0.25%  519.54us         1  519.54us  519.54us  519.54us  cuDeviceTotalMem
                    0.13%  262.87us         1  262.87us  262.87us  262.87us  cudaDeviceSynchronize
                    0.09%  187.58us       101  1.8570us     152ns  77.437us  cuDeviceGetAttribute
                    0.02%  41.571us         1  41.571us  41.571us  41.571us  cuDeviceGetName
                    0.02%  35.077us         1  35.077us  35.077us  35.077us  cudaLaunchKernel
                    0.01%  20.030us         1  20.030us  20.030us  20.030us  cuDeviceGetPCIBusId
                    0.00%  2.0600us         3     686ns     199ns  1.1440us  cuDeviceGetCount
                    0.00%  1.6560us         2     828ns     302ns  1.3540us  cuDeviceGet
                    0.00%     317ns         1     317ns     317ns     317ns  cuDeviceGetUuid
```

De nuevo, nos muestra que no ha habido ningún error. Por último, podemos probar a poner matrices cuyas dimensiones no sean potencias de dos, para reafirmarnos en que la forma de crear el grid y el kernel es correcta para cualesquiera dimensiones:

```
!sed -i '/#define R/c\#define R 111' sumaMatrices.cu
!sed -i '/#define C/c\#define C 333' sumaMatrices.cu
!/usr/local/cuda/bin/nvcc -arch=sm_35 -rdc=true
            sumaMatrices.cu -o sumaMatrices_changedim_2 -lcudadevrt
!nvprof ./sumaMatrices_changedim_2
```

Que nos da como resultado:

```
==792== NVPROF is profiling process 792, command: ./sumaMatrices_changedim_2
==792== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
Número de errores: 0
==792== Profiling application: ./sumaMatrices_changedim_2
==792== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.69%  46.240us         2  23.120us  22.080us  24.160us  [CUDA memcpy HtoD]
                   26.97%  21.248us         1  21.248us  21.248us  21.248us  [CUDA memcpy DtoH]
                   14.34%  11.296us         1  11.296us  11.296us  11.296us  add(int*, int*, int*,
      API calls:   99.39%  207.92ms         3  69.306ms  4.0150us  207.91ms  cudaMalloc
                    0.26%  546.53us         1  546.53us  546.53us  546.53us  cuDeviceTotalMem
                    0.12%  243.93us         3  81.309us  52.364us  130.09us  cudaMemcpy
                    0.12%  240.88us       101  2.3840us     177ns  87.481us  cuDeviceGetAttribute
                    0.07%  144.59us         3  48.196us  3.9960us  120.94us  cudaFree
                    0.02%  38.427us         1  38.427us  38.427us  38.427us  cudaLaunchKernel
                    0.02%  31.407us         1  31.407us  31.407us  31.407us  cuDeviceGetName
                    0.01%  18.388us         1  18.388us  18.388us  18.388us  cudaDeviceSynchronize
                    0.00%  8.4000us         1  8.4000us  8.4000us  8.4000us  cuDeviceGetPCIBusId
                    0.00%  2.7880us         3     929ns     279ns  1.2740us  cuDeviceGetCount
                    0.00%  1.8820us         2     941ns     412ns  1.4700us  cuDeviceGet
                    0.00%     317ns         1     317ns     317ns     317ns  cuDeviceGetUuid
```

Nuevo resultado sin errores. Vemos como en este caso al ser las dimensiones más pequeñas, los tiempos de copia entre CPU y GPU son bastante menores que en los casos anteriores.

1.5. Stencil1d: Estudiar el efecto de la memoria compartida

Ejercicio 4

Compile el siguiente código, explique su funcionamiento y compruebe si la salida es correcta. En caso de que sea necesario realice las modificaciones oportunas.

Analizamos primero el código por partes. Comenzamos con el kernel
`void stenci_1d(int* i, int *out):`

```
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + (blockIdx.x * blockDim.x) + RADIUS;
int lindex = threadIdx.x + RADIUS;
```

Tenemos una declaración de una variable `__shared__`, que será compartida entre las hebras del mismo bloque. Entonces, se obtienen los dos índices usando el índice de la hebra, el del bloque, y un radio que está previamente fijado a 3. En concreto, lo que se hace es coger como `gindex` el índice que está en la posición $j + i + RADIUS$, donde j es el índice de la hebra, i es el número: $bloque * num\_bloques$. Además, tenemos fijado $RADIUS = 3$.

```
// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS){
```

```
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
// Make sure all threads get to this point before proceeding!
    //__syncthreads();
```

Aquí, se actualizan los valores de la variable compartida que habíamos declarado. Además, y más importante, nos encontramos un `__syncthreads` comentado. Esto posiblemente tendremos que modificarlo a continuación para activar la sincronización de las hebras.

```
// Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex-RADIUS] = result;
```

En este último trozo de la función se realiza la operación requerida: **en la posición i de la salida, estará la suma de los elementos desde** $i - RADIUS$ **hasta** $i + RADIUS$. En general, debemos tener cuidado en este programa, pues estamos viendo que habrá posiciones del vector que sean consultadas por varias hebras. Queremos hacer nuestro programa para que cada hebra calcule una posición del vector de salida así que podría haber accesos al vector original en múltiples ocasiones. Para este programa, hay que tener cuidado en la inicialización del vector de entrada:

```
unsigned int i;
int h_in[NUM_ELEMENTS + 2 * RADIUS], h_out[NUM_ELEMENTS];
int *d_in, *d_out;

// Initialize host data
for( i = 0; i < (NUM_ELEMENTS + 2*RADIUS); ++i )
  h_in[i] = 1; // With a value of 1 and RADIUS of 3,
                                  all output values should be 7
```

Como vemos, es importante reservar e inicializar $2*RADIUS$ posiciones más de la cuenta para el vector, pues accederemos a ellas para el cálculo de la posición 0 y $NUM\_ELEMENTS$ de nuestro vector de salida.

1.5.1. Ejecuciones

Ahora, vamos al ejercicio. Se nos pide que estudiemos primero una versión sin memoria compartida. Para ello, reescribimos la función de kernel del siguiente modo para que quede más claro:

```
__global__ void stencil_1d(int *in, int *out)
{
    int index = threadIdx.x + (blockIdx.x * blockDim.x) + RADIUS;

    //__syncthreads();
```

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += in[index + offset];

    // Store the result
    out[index-RADIUS] = result;
}
```

Entonces, compilamos y ejecutamos nuestro programa 1000 veces, guardando la salida para ver cuántas veces fallamos. A priori, estamos haciendo mucho acceso a memoria pero no deberíamos tener fallos:

```
!/usr/local/cuda/bin/nvcc -arch=sm_35 -rdc=true stenciltest.cu -o ./stenciltest -l
!for run in {1..1000}; do ./stenciltest ; done > outputs.txt
```

Escribimos este código en python que nos imprimirá el número de veces que el programa ha fallado, si ha habido algún fallo en la ejecución:

```
with open("outputs.txt",'rb') as text:
  info = text.readlines()
  success = [0 if a == b'SUCCESS!\n' else 1 for a in info]
  s = sum(success)
  print(s)
```

```
0
```

Comprobamos que efectivamente no ha habido ningún error. Imprimamos el profiler del programa:

```
==6355== NVPROF is profiling process 6355, command: ./stenciltest
==6355== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
SUCCESS!
==6355== Profiling application: ./stenciltest
==2876== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   43.63%  5.6960us         1  5.6960us  5.6960us  5.6960us  [CUDA memcpy DtoH]
                   32.84%  4.2880us         1  4.2880us  4.2880us  4.2880us  [CUDA memcpy HtoD]
                   23.53%  3.0720us         1  3.0720us  3.0720us  3.0720us  stencil_1d(int*, int*)
      API calls:   99.59%  187.03ms         2  93.513ms  7.0080us  187.02ms  cudaMalloc
                    0.19%  350.98us         1  350.98us  350.98us  350.98us  cuDeviceTotalMem
                    0.09%  163.67us        97  1.6870us     131ns  78.963us  cuDeviceGetAttribute
                    0.06%  109.61us         2  54.804us  14.246us  95.362us  cudaFree
                    0.04%  81.565us         2  40.782us  35.223us  46.342us  cudaMemcpy
                    0.02%  30.194us         1  30.194us  30.194us  30.194us  cuDeviceGetName
                    0.02%  29.932us         1  29.932us  29.932us  29.932us  cudaLaunchKernel
                    0.00%  3.0750us         1  3.0750us  3.0750us  3.0750us  cuDeviceGetPCIBusId
                    0.00%  1.8080us         3     602ns     148ns  1.2980us  cuDeviceGetCount
                    0.00%  1.0740us         2     537ns     269ns     805ns  cuDeviceGet
                    0.00%     251ns         1     251ns     251ns     251ns  cuDeviceGetUuid
```

Vemos que tenemos un 43,6 % del tiempo empleado en copiar de CPU a GPU, y un 33 % del tiempo empleado en copiar de GPU a CPU, lo que suma un 76 % del tiempo solo en copiar datos mientras que solo un 23,5 % del tiempo se emplea en realizar la operación.

Ejercicio 5

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

Como acabamos de mencionar en el comentario anterior, usando el profiler hemos visto que la mayoría del tiempo se ha pasado copiando datos a GPU, pues teníamos que copiarlos en múltiples ocasiones para evitar accesos simultáneos. Por tanto, es en la **copia de datos desde host a GPU** donde tenemos el **cuello de botella**.
Por ello, la modificación que debemos hacer a nuestro programa es tratar de introducir la memoria compartida en nuestro kernel para que este proceso sea más eficiente.

Ejercicio 6

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

Debemos ahora tomar el código explicado inicialmente, pues es en el que declaramos la variable compartida (que tiene `__shared__` delante del tipo) que utilizaremos para hacer uso de esta memoria compartida. Primero, **NO** descomentamos la línea que tiene el `syncthreads`, para ver cómo se comporta nuestor programa cuando no realizamos esta sincronización.
Lo que hacemos es de nuevo compilar el programa y ejecutarlo 100 veces, mostrando cuántas veces obtenemos un resultado erróneo usando el mismo código en python que en el caso anterior:

```
!/usr/local/cuda/bin/nvcc -arch=sm_35 -rdc=true stencilshared.cu -o
                    ./stencilshared -lcudadevrt
!for run in {1..1000}; do ./stencilshared ; done > outputs_shared.txt
```

```
with open("outputs_shared.txt",'rb') as text:
  info = text.readlines()
  success = [0 if a == b'SUCCESS!\n' else 1 for a in info]
  s = sum( success)
  print(s)
```

```
1
```

Lo que está ocurriendo aquí es el fenómeno conocido como **data race**. Este fenómeno ocurre cuando que dos hebras intentan acceder a una de las posiciones a la vez y una intenta escribir sobre la misma y otra leerla, por lo que podría haber valores diferentes a los esperados y por tanto se pueden dar situaciones en las que no obtuviésemos el resultado esperado.
Sabemos que este fallo es aleatorio (pues se produce cuando las hebras no actúan en el orden correcto) y, de hecho, si seguimos ejecutando en la misma sesión del cuaderno de Google Colaboratory puede que no obtengamos más fallos de este estilo. Hemos detectado que este error es más fácilmente reproducible cuando reiniciamos la sesión de ejecución del cuaderno de Colaboratory y ejecutamos todo de nuevo.
Además, sabemos que esto ocurre por los accesos a las posiciones del vector, por lo que si aumentamos el tamaño del vector estamos aumentando la probabilidad de obtener fallos. Hacemos una prueba para comprobar esto. Lo primero que cambiamos es el número de elementos del vector, haciendo que sea $4096 * 8$ (antes, era $4096 * 2$):

```
#define NUM_ELEMENTS  (4096*8)
```

Entonces, volvemos a compilar y ejecutar 1000 veces el código.

```
!/usr/local/cuda/bin/nvcc -arch=sm_35 -rdc=true stencilshared.cu -o
            ./stencilshared -lcudadevrt
!for run in {1..1000}; do ./stencilshared ; done > outputs_shared.txt

with open("outputs_shared.txt",'rb') as text:
  info = text.readlines()
  success = [0 if a == b'SUCCESS!\n' else 1 for a in info]
  s = sum( success)
  print(s)
```

```
7
```

Podemos ver que el número de errores ahora ha aumentado, como (aunque no seguro porque recordamos que es un proceso aleatorio) podía esperarse.
Además, podemos ver utilizando el profiler el siguiente resultado:

```
==2926== Profiling application: ./stencilshared
==2926== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   37.69%  469.57us         1  4.6950us  4.6950us  4.6950us  [CUDA memcpy DtoH]
                   37.18%  463.27us         2  4.2560us  4.2560us  4.2560us  [CUDA memcpy HtoD]
                   25.13%  313.18us         1  3.2000us  3.2000us  3.2000us  stencil_1d(int*, int*)
      API calls:   99.46%  197.26ms         2  98.628ms  4.7020us  197.25ms  cudaMalloc
                    0.28%  559.56us         1  559.56us  559.56us  559.56us  cuDeviceTotalMem
                    0.10%  197.31us       101  1.9530us     165ns  82.106us  cuDeviceGetAttribute
                    0.08%  159.95us         2  79.975us  8.3610us  151.59us  cudaFree
                    0.05%  89.868us         3  29.956us  26.482us  34.831us  cudaMemcpy
                    0.01%  28.654us         1  28.654us  28.654us  28.654us  cuDeviceGetName
                    0.01%  24.893us         1  24.893us  24.893us  24.893us  cudaLaunchKernel
                    0.00%  7.7680us         1  7.7680us  7.7680us  7.7680us  cuDeviceGetPCIBusId
                    0.00%  2.1350us         3     711ns     221ns  1.0340us  cuDeviceGetCount
                    0.00%  1.6970us         2     848ns     302ns  1.3950us  cuDeviceGet
                    0.00%     306ns         1     306ns     306ns     306ns  cuDeviceGetUuid
```

Como vemos, hemos conseguido reducir el tiempo de copia de datos de CPU a GPU en aproximadamente $1us$. Esto era una de las partes que queríamos obtener. Nos faltaría añadir la sincronización de las hebras que hemos comentado para que además de reducir el cuello de botella no tengamos ningún fallo de cálculo.

Ejercicio 7

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

Para esta última parte, bastaría con descomentar la línea que tiene esta función para obtener la sincronización en ese punto del código de todas las hebras. Vamos a añadir también al código un bucle `for` que ejecute el mismo código 100 veces desde dentro del programa, para ver si mantenemos la mejora en tiempo y además obtenemos los resultados correctos en nuestro programa. Ejecutamos el profiler y obtenemos el siguiente resultado

```
==3079== NVPROF is profiling process 3079, command: ./stencilshared_sync
Número de errores: 0
==3079== Profiling application: ./stencilshared_sync
==3079== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   37.58%  470.82us       100  4.7080us  4.4160us  6.0480us  [CUDA memcpy DtoH]
                   36.93%  462.63us       100  4.6260us  4.2880us  6.1760us  [CUDA memcpy HtoD]
                   25.49%  319.27us       100  3.1920us  3.1360us  3.2960us  stencil_1d(int*, int*)
      API calls:   73.46%  15.0458s       200  75.229ms  6.2800us  187.42ms  cudaMalloc
                   26.45%  5.41661s       100  54.166ms  52.283ms  58.709ms  cudaDeviceReset
                    0.05%  9.5939ms       200  47.969us  8.9190us  127.93us  cudaFree
                    0.03%  6.1591ms       200  30.795us  24.055us  52.484us  cudaMemcpy
                    0.01%  2.8672ms       100  28.671us  24.738us  50.801us  cudaLaunchKernel
                    0.00%  372.52us         1  372.52us  372.52us  372.52us  cuDeviceTotalMem
                    0.00%  155.56us       101  1.6030us     135ns  65.806us  cuDeviceGetAttribute
                    0.00%  29.607us         1  29.607us  29.607us  29.607us  cuDeviceGetName
                    0.00%  3.7980us         1  3.7980us  3.7980us  3.7980us  cuDeviceGetPCIBusId
                    0.00%  2.1450us         3     715ns     139ns  1.4620us  cuDeviceGetCount
                    0.00%  1.2110us         2     605ns     259ns     952ns  cuDeviceGet
                    0.00%     367ns         1     367ns     367ns     367ns  cuDeviceGetUuid
```

Los resultados son exitosos. Se consigue mantener la reducción de tiempo en media respecto a la versión que no tenía memoria compartida, además de obtener ahora un código sin errores. Podríamos decir por tanto que hemos obtenido una aceleración de $4{,}7/5{,}7 \approx 20\,\%$.

2. Parte 2: Programación con QisKit: Computación cuántica

Para esta parte de la práctica utilizaremos el Quantum Composer de IBM. Puesto que el tenemos un número limitado de procesos a ejecutar (únicamente 5) veremos los resultados en el simulador sin llegar a medirlo en muchos casos.

2.1. Puertas Cuánticas

Ejercicio 8

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

Puerta NOT

La única operación no trivial aplicable sobre un único bit es la negación: la puerta NOT. De la misma forma, es natural preguntarse cuál es el equivalente a la puerta NOT en el mundo cuántico. Dado que un qubit está descrito por dos amplitudes α y β:

$$|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

la puerta NOT será un intercambio entre las posiciones de estas amplitudes, obteniéndose así:

$$|\varphi\rangle = \beta|0\rangle + \alpha|1\rangle.$$

La matriz unitaria que describe esta transformación es sencilla:

$$X = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Vemos la implementación de esta puerta en el Quantum Composer de IBM:

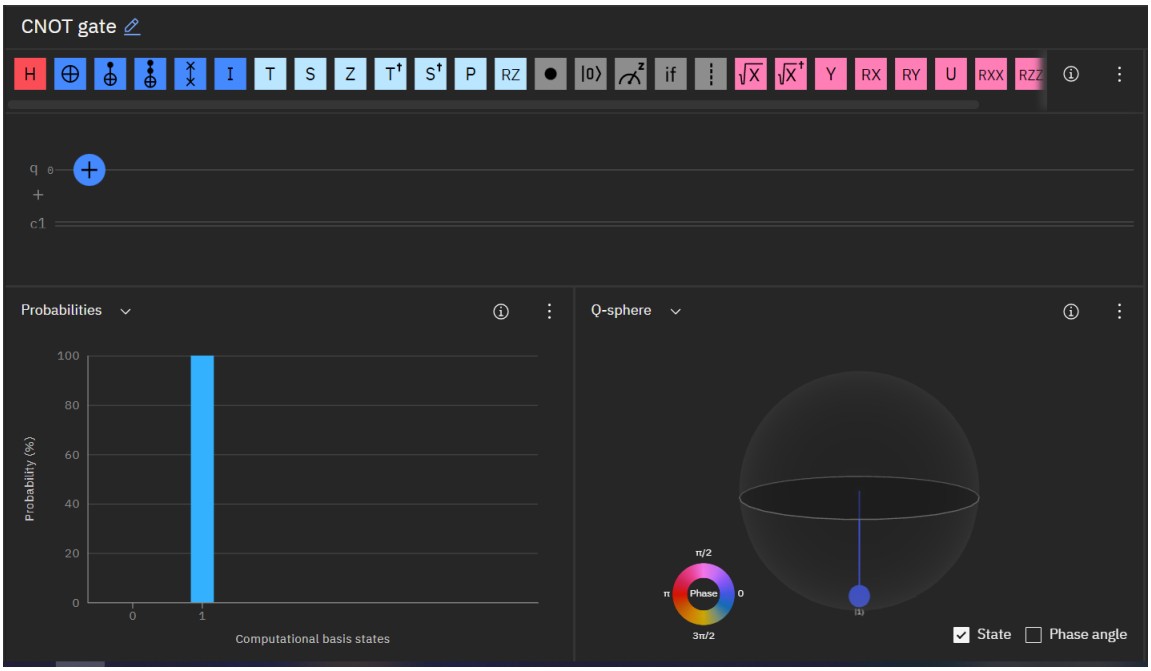


Figura 1: Circuito con puerta de NOT

Recordemos que en el Quantum Composer de IBM todos los qubits empiezan siempre en el estado $|0\rangle$. Tras aplicarlo a nuestro qubit una puerta X obtendremos $|1\rangle$.

Puerta Hadamard

Finalmente presentamos la puerta de Hadamard para un único bit. Está descrita por la siguiente matriz unitaria:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Uno de sus usos más comunes es la superposición de qubits. Si aplicamos esta puerta al estado $|0\rangle$ obtenemos el estado de Bell:

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = |+\rangle$$

Mientras que si se la aplicamos al estado $|1\rangle$ obtenemos:

$$H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle$$

Que también supone una superposición exacta de $|0\rangle$ y $|1\rangle$ puesto que $|1/\sqrt{2}|^2 = |-1/\sqrt{2}|^2 = 1/2$.
Podemos estudiar el comportamiento de esta puerta utilizando el Quantum Composer de IBM:

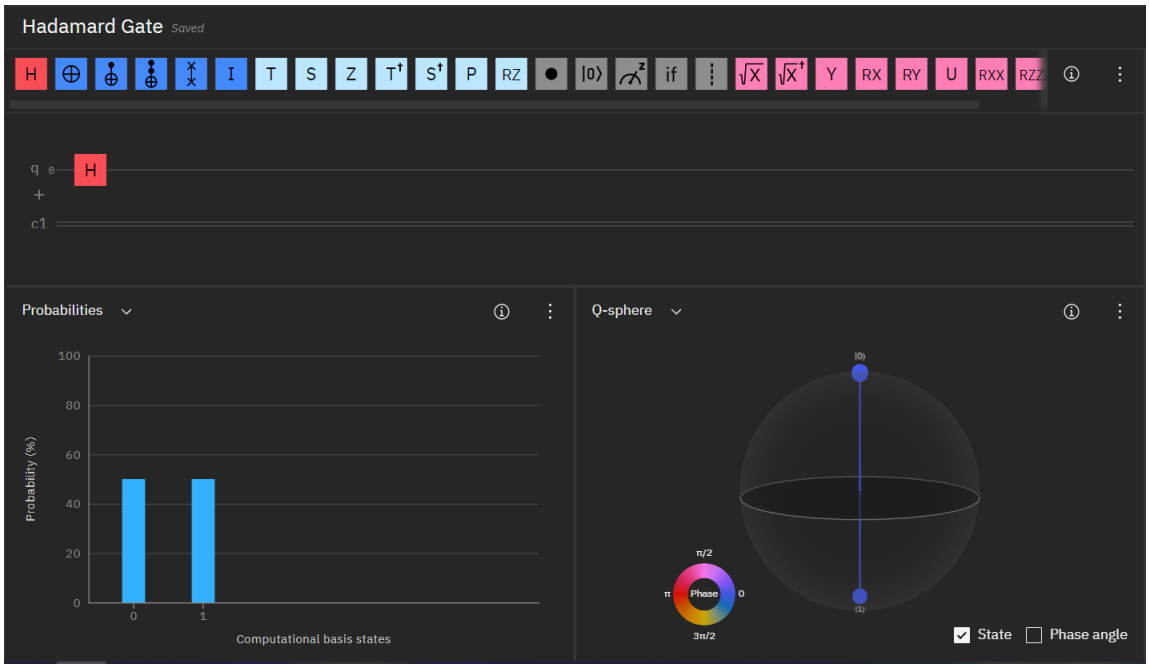


Figura 2: Circuito con puerta de Hadamard

Mirando tanto la esfera de Bloch como las probabilidades vemos que tenemos la misma probabilidad de medir 0 y 1.

Puertas de Pauli

Un conjunto particularmente relevante de puertas son las descritas por las matrices de Pauli:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}; \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Ya conocemos la matriz X, descrita también como la puerta NOT quántica.

Puertas de fase

La puerta $\pi/8$, normalmente descrita por la letra T, es una *puerta de fase*. Las puertas de fase son un tipo especial de puertas cuánticas que llevan $|0\rangle \mapsto |0\rangle$ y $|1\rangle \mapsto e^{i\phi}|1\rangle$, donde ϕ es un ángulo de giro. El término $e^{i\phi}$ se denomina *fase* y no afecta a los resultados de las mediciones 0 y 1. En particular, la puerta T cumple $\phi = \pi/4$, y la puerta Z de Pauli es una puerta fase con $\phi = \pi/2$:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}; \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} = -1 \end{pmatrix}$$

En computación clásica, uno de los resultados básicos más relevante es que cualquier función booleana puede describirse utilizando únicamente las puertas clásicas AND, OR y NOT. De la misma forma, en computación cuántica se obtiene siguiente resultado

Teorema. *Toda matriz unitaria puede aproximarse con una combinación de puertas Hadamard, CNOT y $\pi/8$.*

Esto es, todo circuito cuántico puede describirse utilizando únicamente dichas puertas.

Puerta CNOT

Pasamos a estudiar la única puerta de dos qubits que comentaremos en este apartado y que ya ha aparecido en la explicación anterior: la puerta Controlled-NOT, o simplemente CNOT. Esta transformación queda descrita por la siguiente matriz unitaria:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Esta puerta mapea $|q_1\rangle|q_0\rangle \mapsto |q_1 \oplus q_0\rangle|q_0\rangle$. Vemos su funcionamiento.

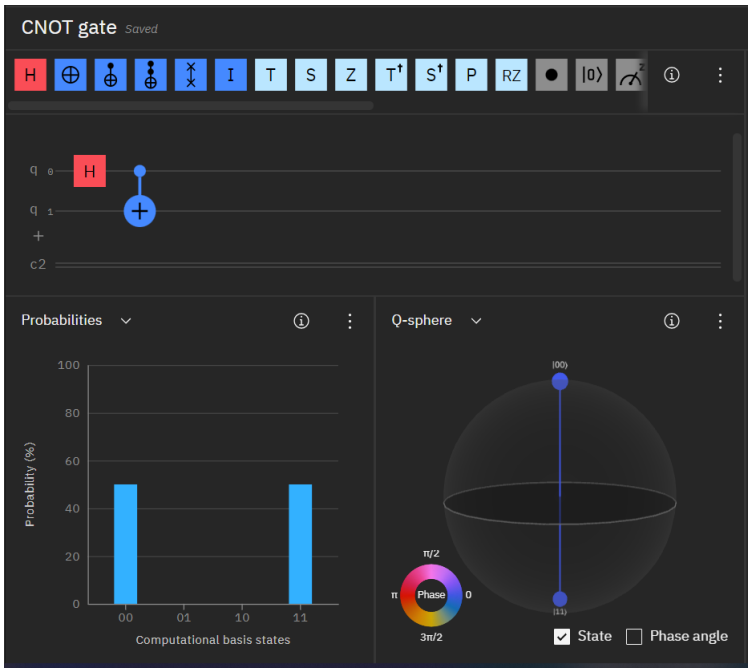


Figura 3: Circuito con puerta CNOT

Si le aplicamos la puerta CNOT a $|+\rangle|0\rangle$ obtenemos el estado $(|00\rangle + |11\rangle)/\sqrt{2}$. Esto es: si el estado q_0 era inicialmente un 0, no cambiará el estado q_1. Por otro lado, si el estado q_0 era

inicialmente un 1, el qubit q_1 cambiará a 1. Explicaremos este circuito en profundidad en la sección 2.3.

Puerta Toffoli

Finalmente, explicamos una única puerta de más de dos qubits: la puerta de Toffoli o puerta CCNOT. Esta puerta será necesaria para el sumador de dos qubits de la sección 2.4 y será sencilla de entender una vez comprendida la puerta anterior. Queda descrita por la siguiente matriz unitaria:

$$CCNOT = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Y mapea $|q_2\rangle|q_1\rangle|q_0\rangle \mapsto |q_2 \oplus (q_0 \wedge q_1)\rangle|q_1\rangle|q_0\rangle$. Es decir, altera el último qubit si y sólo si los dos primeros son 1. Veremos un ejemplo de uso de esta puerta en la sección 2.4.

2.2. Generación de números aleatorios con un Computador Cuántico

Ejercicio 9

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.

Sabemos que utilizando la puerta de Hadarmad H explicada en el apartado anterior ponemos un qubit $|0\rangle$ en superposición:

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Si ahora medimos este qubit obtendremos $|0\rangle$ con probabilidad $|1/\sqrt{2}|^2 = 1/2$, y $|1\rangle$ con probabilidad $1/2$. Esto es, hemos creado un generador de bits aleatorios utilizando un único qubit. Para crear un generador de 3 bits utilizaremos un sistema de 3 qubits. Inicialmente en el estado $|000\rangle$, aplicaremos una puerta Hadamard a cada qubit de forma independiente, poniendo así cada qubit en superposición:

$$\hat{H}_8|00000000\rangle = \frac{1}{\sqrt{2^8}}\big(|0000000\rangle + |00000001\rangle + \ldots + |11111111\rangle\big)$$

Donde la puerta H_8 tranformación de Hadamard para ocho qubits. Se puede definir recursivamente de la siguiente forma:

$$H_m = H_1 \otimes H_{m-1}; \quad H_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Pasamos a realizar un estudio empírico del circuito diseñado. Lo implementamos utilizando el Quantum Composer de IBM:

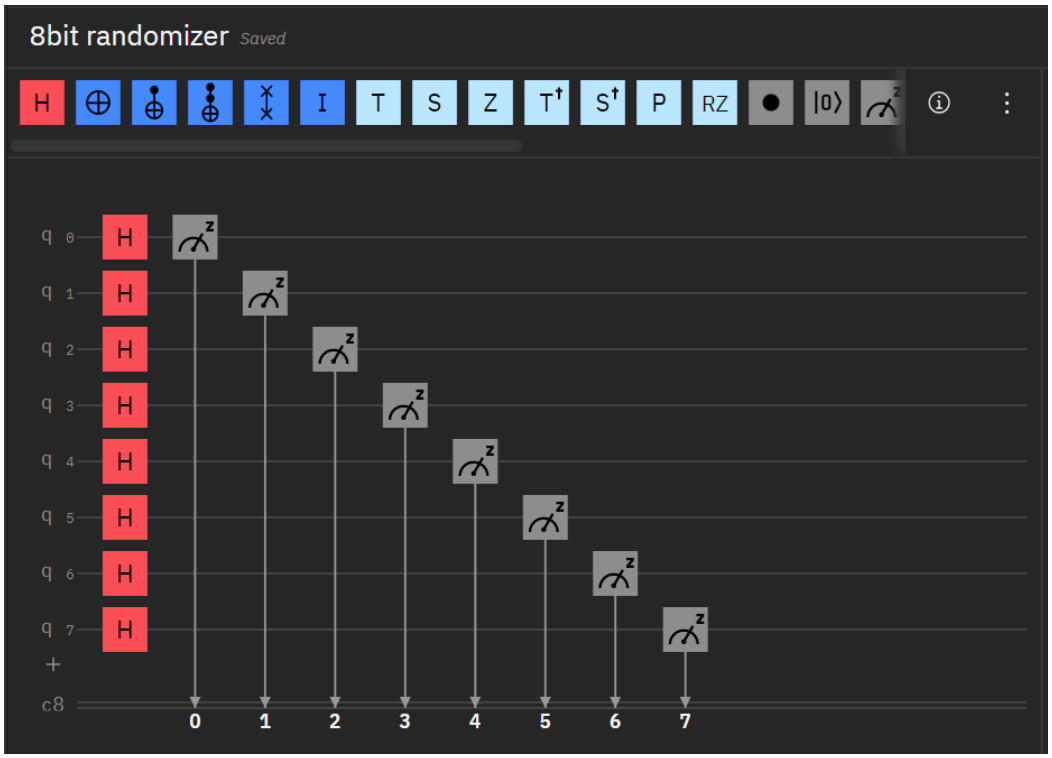


Figura 4: Circuito randomizador de 8 bits.

Este simple circuito cuántico pone todos los qubits en superposición y después mide el resultado. Ejecutamos el experimento en el simulador del ordenador cuántico de IBM, obteniendo los siguientes resultados:

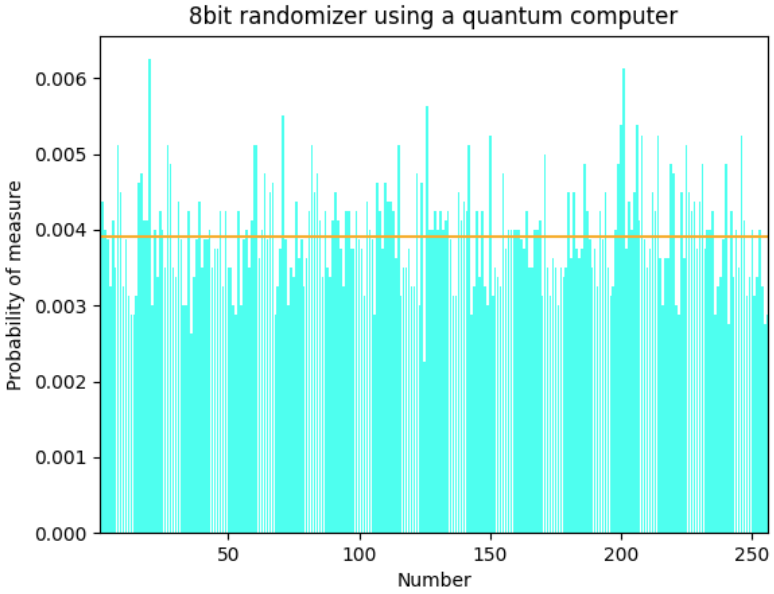


Figura 5: Resultados de la simulación con 8000 ejecuciones

Estas mediciones han de aproximarse a una uniforme de parámetro $1/256$. Podemos apreciar en la gráfica como los resultados son cercanos a este valor pero distan mucho de definir claramente una uniforme. Podemos comparar estos resultados con la generación de números aleatorios utilizando *scipy* en nuestra propia máquina:

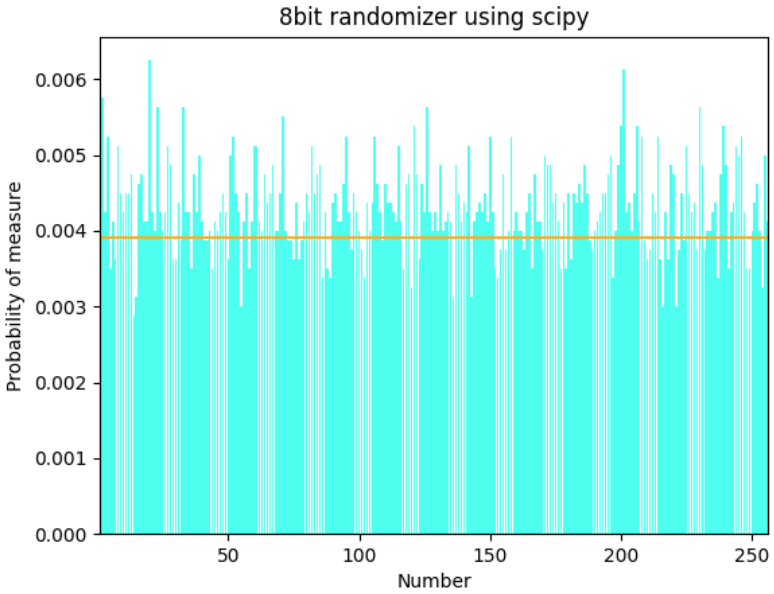


Figura 6: Resultados de la simulación con 8000 ejecuciones

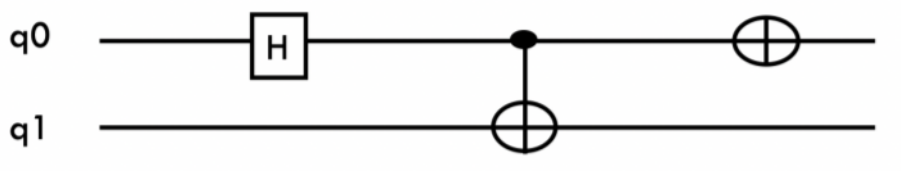
Comparando ambas gráficas podemos apreciar como la generación de números se acerca a

la distribución uniforme mencionada pero en ambas estamos relativamente lejos del modelo teórico. Tras ver los resultados de esta generación utilizando *scipy* podemos asegurar que el generador utilizando el Quantum Composer de IBM obtiene números aleatorios razonablemente aleatorizados.

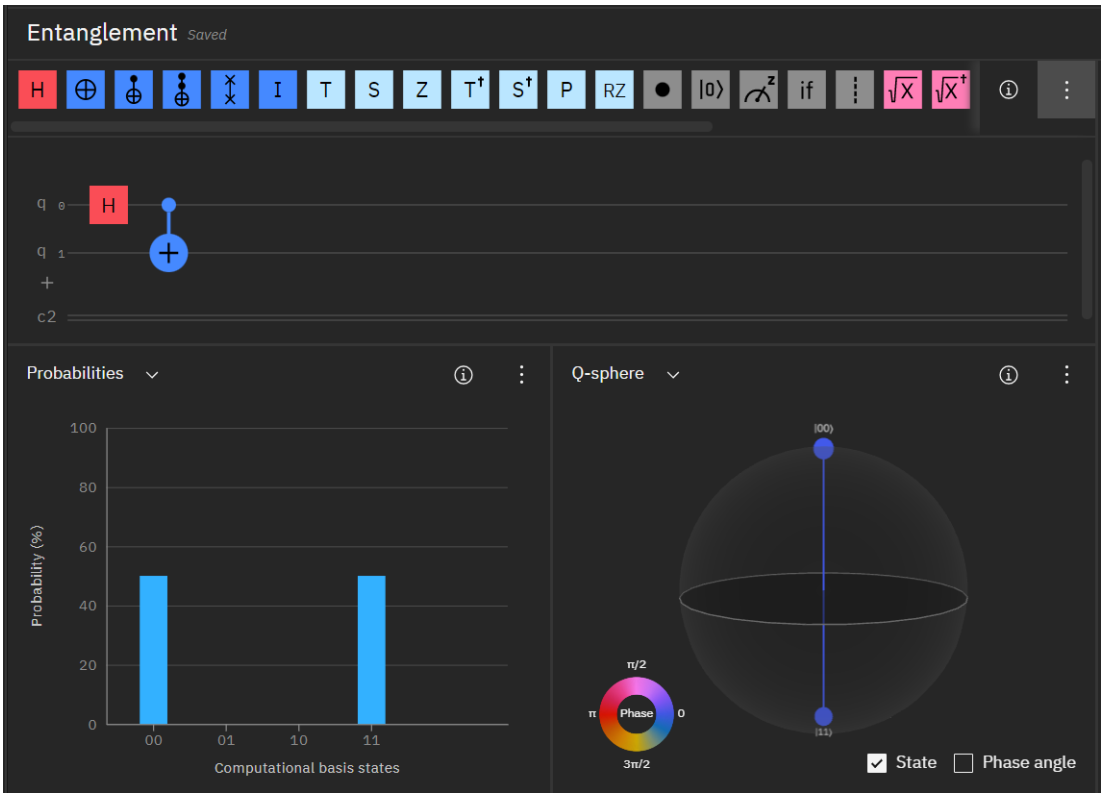
2.3. Entrelazamiento

Ejercicio 10

Explica el funcionamiento del siguiente circuito cuántico:



Comenzaremos estudiando un circuito ligeramente más sencillo que también produce entrelazamiento cuántico:



Conociendo ya la puerta de Hadamard, sabemos que el resultado tras la aplicación de dicha puerta al estado $|0\rangle$ será $|+\rangle$. Utilizando a continuación la puerta CNOT, como el estado de

control tiene la misma probabilidad de ser $|0\rangle$ que $|1\rangle$, el segundo qubit tendrá la misma probabilidad de tener dichos valores. Analíticamente:

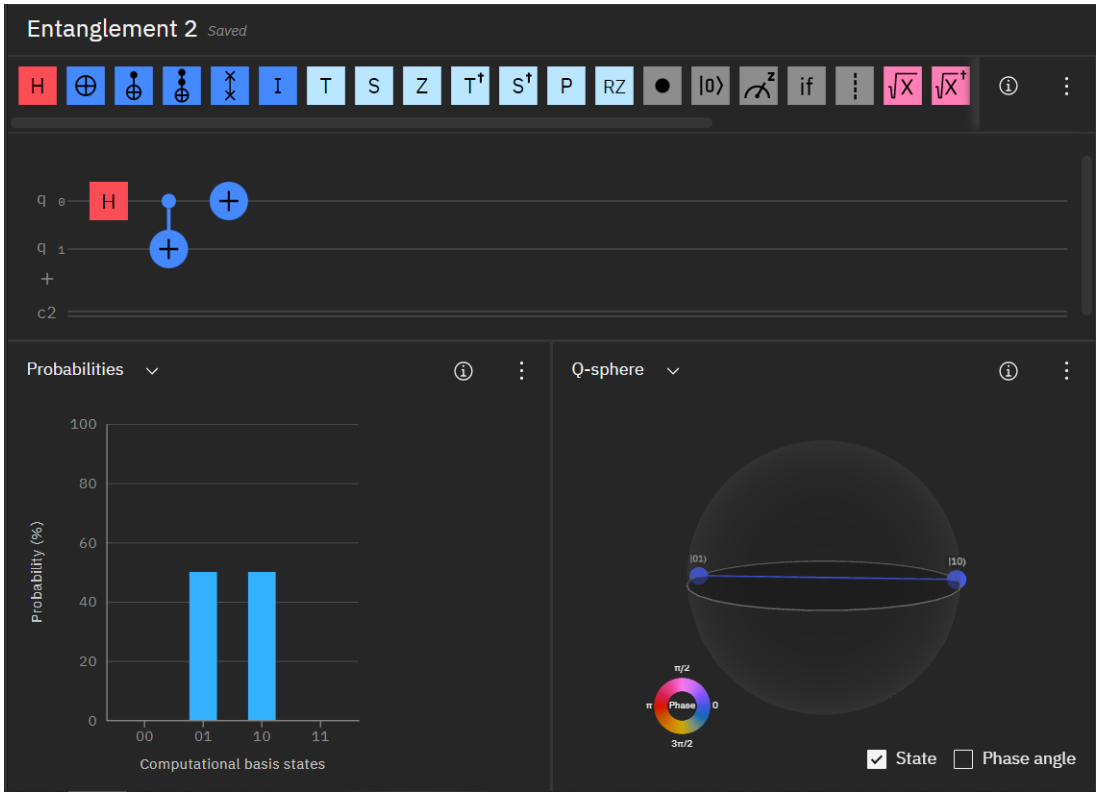
$$|+\rangle|0\rangle CNOT = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\Phi^+\rangle$$

Obteniéndo así el famoso estado EPR (Einstein, Podolsky y Rosen) o estado de Bell $|\Phi^+\rangle$. En este aparentemente sencillo estado puede apreciarse el entrelezamiento cuántico, lo que dió lugar a dicha paradoja. Estudiémos este estado en detalle.

Para empezar, al medir ambos qubits únicamente podremos obtener los resultados 00 y 11. Si únicamente medimos uno de los dos qubits y obtenemos, por ejemplo, un 0, entonces cuando midamos el otro estado obtendremos otro 0 con toda probabilidad, pues los únicos resultados finales válidos son los anteriormente descritos 00 y 11. Lo mismo ocurre si medimos 1: el valor del segundo qubit ha de ser también un 1.

De esta forma, hemos hecho que el segundo qubit colapse a un estado al medir otro qubit distinto. Estas son las implicaciones del entrelazamiento cuántico.

Estudiémos ahora el circuito del enunciado:



Esto es, aplicarle una puerta NOT al primer qubit del estado de Bell, obteniendo:

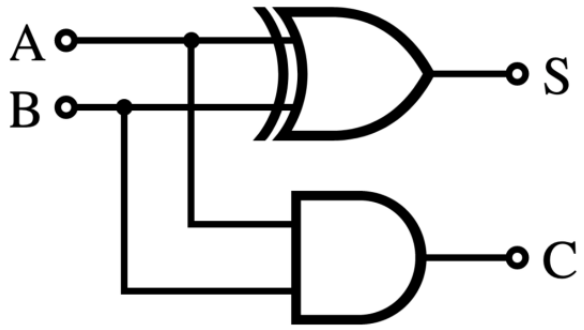
$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}(X \otimes \mathbb{I}) = \frac{|10\rangle + |01\rangle}{\sqrt{2}}$$

Este estado se comporta como el anterior pero con valores de medida opuestos: si medimos un 0 en el primer qubit, el segundo colapsará automáticamente al estado 1, y viceversa.

2.4. Sumador de 2 qbits

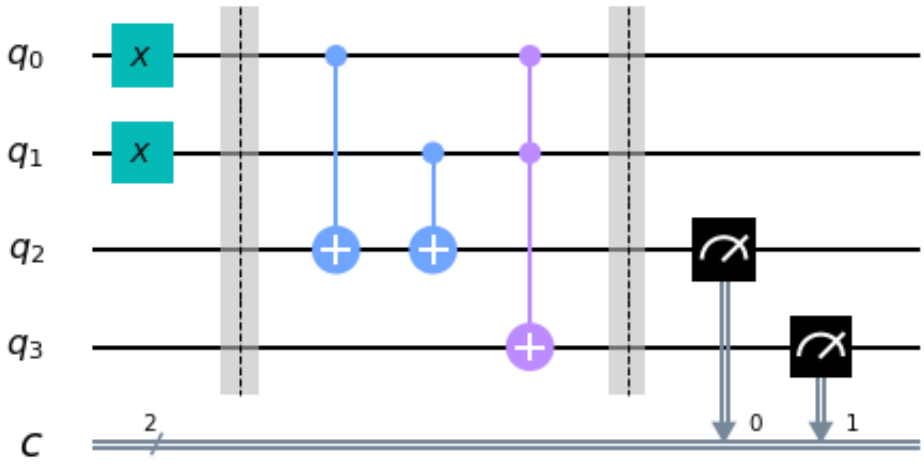
Ejercicio 11

Realice el circuito cuántico que equivale a un sumador de dos bits equivalente al siguiente circuito clásico:



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 este enlace.

El circuito cuántico que implementa el sumador de 2 qubits es el siguiente, donde los qubits de entrada son q_0, q_1 y q_2, q_3 son los qubits de salida:



Entendámoslo paso a paso. Puesto que los qubits q_0, q_1 son de entrada, los valores que aparecen

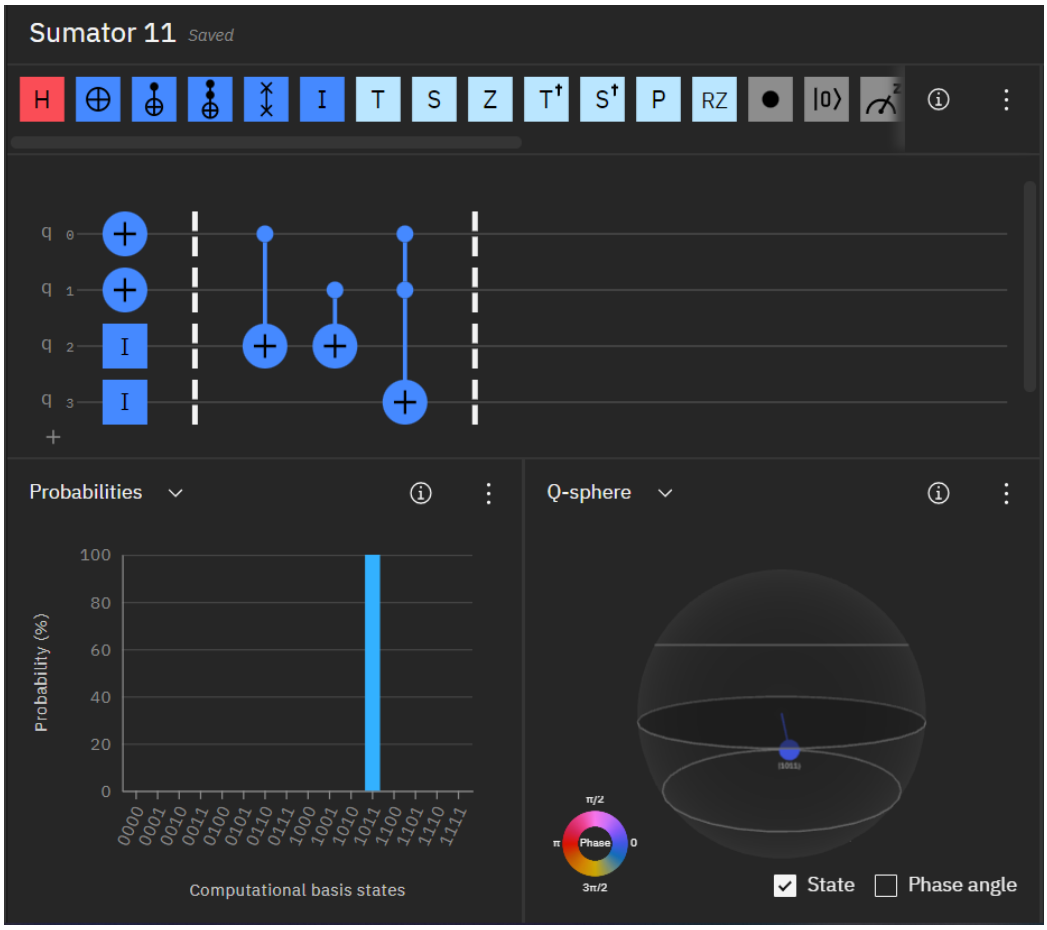
antes de la primera línea puntea ($q_0, q_1 = |1\rangle$) son un ejemplo en este caso.

Como vimos en ejercicios anteriores, la primera puerta CNOT mapea $|q_2\rangle|q_0\rangle \mapsto |q_2\rangle|q_0 \oplus q_2\rangle$. Puesto que inicialmente $|q_2\rangle = |0\rangle$, tras aplicar la primera puerta obtenemos $|0\rangle|q_0\rangle \mapsto |0 \oplus q_0\rangle|q_0\rangle = |q_0\rangle|q_0\rangle$.

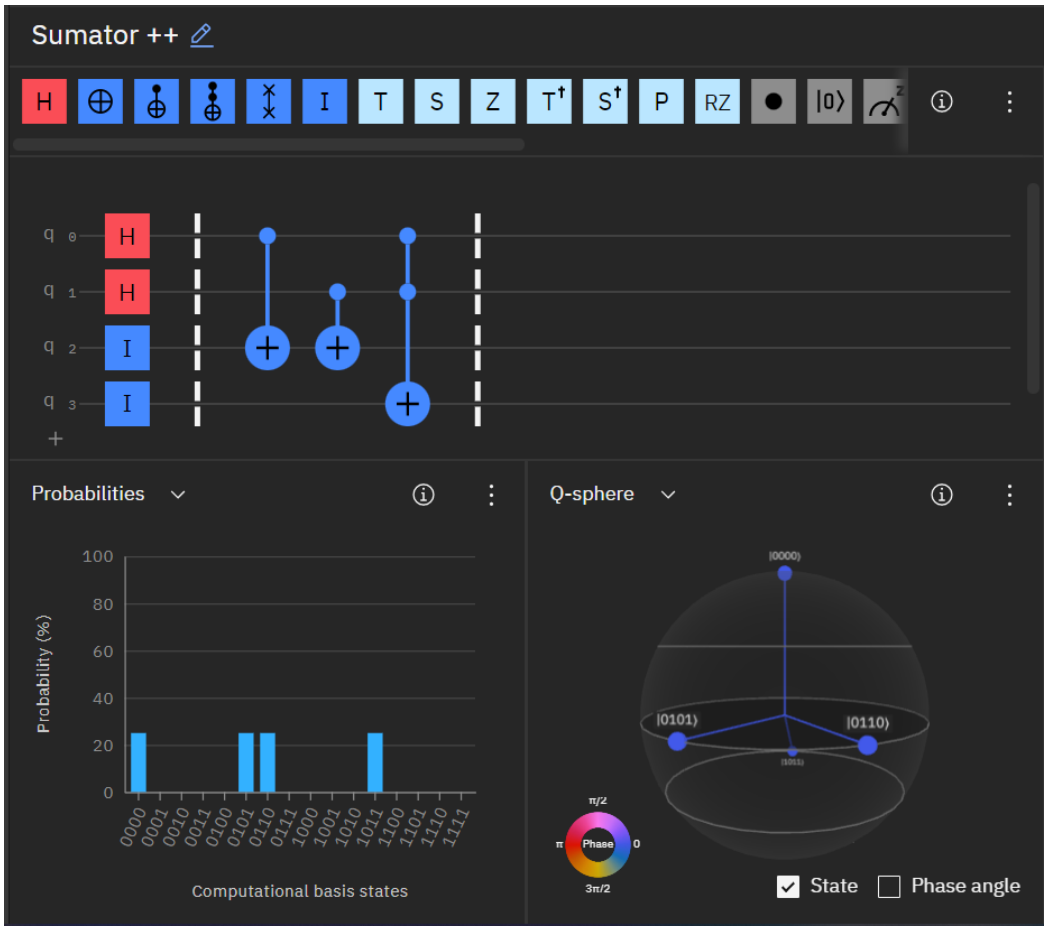
Al aplicar la segunda puerta X obtenemos en el tercer qubit $|q_0 \oplus q_1\rangle$. Esto completa la primera parte de nuestro sumador de dos qubits.

De cara al acarreo acumulado en q_3, este unicamente es 1 si ambos qubits son 1. Por lo tanto, utilizamos la puerta de Toffoli para mapear $|0\rangle|q_1\rangle|q_0\rangle \mapsto |0 \oplus (q_0 \wedge q_1)\rangle|q_1\rangle|q_0\rangle = |q_0 \wedge q_1\rangle|q_1\rangle|q_0\rangle$.

Comprobemos el resultado de este circuito utilizando el Quantum Composer de IBM para distintas entradas. En primer lugar utilizamos $|11\rangle$ para los qubits de entrada como en el circuito proporcionado:



Podemos ver en las probabilidades de las salidas como 1011 es el único resultado posible (a nivel teórico) de nuestro circuito. Recordemos que esto representa las medidas de $q_3 q_2 q_1 q_0$, en ese orden. Es decir, tras sumar dos estados $|1\rangle$ obtenemos $|10\rangle$, como cabría esperar. Estudiémos ahora el resultado de nuestro circuito para dos entradas $|+\rangle$:



Obtenemos las siguientes salidas, con igual probabilidad:

- $\mathrm{P}(q_3q_2q_1q_0 = 0000) = 0{,}25$.
- $\mathrm{P}(q_3q_2q_1q_0 = 0101) = 0{,}25$.
- $\mathrm{P}(q_3q_2q_1q_0 = 0110) = 0{,}25$.
- $\mathrm{P}(q_3q_2q_1q_0 = 1011) = 0{,}25$.

Sin embargo, si miramos unicamente los qubits de salida obtenemos la siguiente distribución:

- $\mathrm{P}(q_3q_2 = 00) = 0{,}25$.
- $\mathrm{P}(q_3q_2 = 01) = 0{,}50$.
- $\mathrm{P}(q_3q_2 = 10) = 0{,}25$.

Lo que concuerda con las predicciones teóricas: la probabilidad de que ambos qubits iniciales sean 0 o 1 es $0{,}25$, mientras que la probabilidad de que uno sea 0 y el otro 1 es de $0{,}5$.

3. Ejercicios opcionales: estudio de algoritmos cuánticos

Para la parte opcional de esta práctica explicaremos en detalle el funcionamiento de algunos algoritmos cuánticos.

3.1. Teleportación cuántica

El objetivo de este algoritmo es el de transmitir un qubit entre dos personas llamadas Alice y Bob. Antes de realizar el experimento, Bob y Alice comparten un qubit entrelazado. Aunque este proceso es equivalente para cualquier otro estado de la base de Bell, tomaremos el estado de Bell presentado en la sección 2.3:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{|0_A\rangle \otimes |0_B\rangle + |1_A\rangle \otimes |1_B\rangle}{\sqrt{2}}$$

Donde utilizamos la notación convencional de la física cuántica:

$$|0_A\rangle \otimes |1_B\rangle \equiv |0_A\rangle|1_B\rangle \equiv |01\rangle$$

Una vez el estado entrelazado ha sido correctamente creado, Alice y Bob toman cada uno de los qubits entrelazados y se separan una distancia suficiente para que ambos sistemas no interfieran entre ellos. La distancia entre Alice y Bob peude ser tan grande como se quiera mientras los qubits permanezcan entrelazados (no haya ninguna alteración a los qubits en su transporte).

El objetivo será transmitir un nuevo qubit $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$ de Alice a Bob. Para ello asumimos que las amplitudes α y β son desconocidas para ambos. De esta forma, nuestro sistema inicial consta de tres qubits: el primero pertenece a Alice y procede del estado $|\Psi\rangle$, el segundo también pertenece a Alice y procede del estado compartido y entrelazado $|\Phi^+\rangle$, y el tercero que pertenece a Bob y también procede del estado compartido $|\Phi^+\rangle$. Podemos escribir el estado inicial de nuestro sistema completo de la siguiente forma:

$$|\Psi_0\rangle = |\Psi\rangle|\Phi^+\rangle = |00\rangle + |11\rangle = (\alpha|0\rangle + \beta|1\rangle)\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|00\rangle + |11\rangle)}{\sqrt{2}}$$

De nuevo, recordemos que el objetivo es enviar el estado $|\Psi\rangle$ de Alice a Bob. Para ello tendremos que, además de aplicar nuestro circuito cuántico al sistema, enviar dos únicos bits clásicos de Alice a Bob. Estudiémos el circuito paso a paso:

Primer paso: Puerta CNOT

El primer paso de nuestro algoritmo es aplicar una puerta CNOT a los dos qubits de Alice. Esto es, aplicar al sistema completo la transformación $CNOT \otimes \mathbb{I}$:

$$|\Psi_1\rangle = (CNOT \otimes \mathbb{I})|\Psi_0\rangle = \frac{\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|10\rangle + |01\rangle)}{\sqrt{2}}$$

De esta forma únicamente habrá cambiado el segundo qubit de Alice en el sistema - correspondiente al qubit central de nuestro sistema completo.

Segundo paso: Puerta de Hadamard

Aplicamos la puerta de Hadamard al qubit de Alice, obteniendo:

$$|\Psi_2\rangle = (H \otimes \mathbb{I} \otimes \mathbb{I})|\Psi_1\rangle = \frac{\alpha|+\rangle(|00\rangle + |11\rangle) + \beta|-\rangle(|10\rangle + |01\rangle)}{\sqrt{2}}$$
$$= \frac{1}{\sqrt{2}}\left(\frac{\alpha}{\sqrt{2}}(|0\rangle + |1\rangle)|(|00\rangle + |11\rangle) + \frac{\beta}{\sqrt{2}}(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)\right)$$

Operamos en la expresión anterior para agrupar los qubits de Alice:

$$|\Psi_2\rangle = \frac{1}{2}\Big(|00\rangle_A(\alpha|0\rangle_B + \beta|1\rangle_B) + |01\rangle_A(\alpha|1\rangle_B + \beta|0\rangle_B)$$
$$+ |10\rangle_A(\alpha|0\rangle_B - \beta|1\rangle_B + |11\rangle_A(\alpha|1\rangle_B - \beta|0\rangle_B\Big)$$

Último paso: Medida y transmisión

Como podemos ver en la expresión anterior de $|\Psi_2\rangle$, para cada posible valor de los qubits de Alice tenemos un estado distinto para el qubits de Bob. El resto del proceso consiste en medir los estados de Alice y transmitir la medida a Bob. Por ejemplo, si Alice mide en sus dos qubits el valor 00, el qubit de Bob estará en el estado $\alpha|0\rangle_B + \beta|1\rangle_B = |\Psi\rangle$.

Si el resultado de la medida de Alice fuese distinto a 00, Bob no tendría el estado $|\Psi\rangle$ como tal, pero podría recuperarlo aplicando operaciones sencillas. Para saber qué puertas ha de aplicarle a su qubit necesita saber cúal es su contenido actual. Es por ello que necesita la información de la medida transmitida por Alice.

En la siguiente tabla podemos ver un desglose de las operaciones que ha de aplicar Bob según la medida de Alice para recuperar el qubit original.

Medida de Alice m	**Estado de Bob** $\|\Psi_3\rangle_m$	**Operación a aplicar**	**Resultado**
00	$\alpha\|0\rangle + \beta\|1\rangle$	$\mathbb{I}$	$\mathbb{I}\|\Psi_3\rangle_{00} = \|\Psi\rangle$
01	$\alpha\|1\rangle + \beta\|0\rangle$	X	$X\|\Psi_3\rangle_{01} = \|\Psi\rangle$
10	$\alpha\|0\rangle - \beta\|1\rangle$	Z	$Z\|\Psi_3\rangle_{10} = \|\Psi\rangle$
11	$\alpha\|1\rangle - \beta\|0\rangle$	XZ	$XZ\|\Psi_3\rangle_{11} = \|\Psi\rangle$

En la figura 7 podemos ver el circuito cuántico que implementa este algoritmo, donde la puerta morada es la *Controlled-Z*, que aplica la puerta Z al tercer qubit si y sólo si el estado del primer qubit es 1.

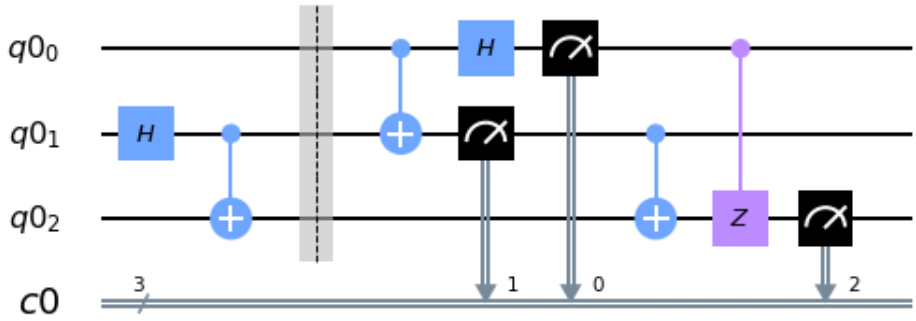


Figura 7: Circuito cuántico que implementa la teleportación cuántica.

Finalmente, merece la pena comentar la paradoja que surgió a partir de este circuito cuántico. Ya que Alice y Bob están tan separados físicamente como se quiera, al medir los qubits de Alice estamos haciendo el qubit de Bob (a distancia arbitrariamente grande) colapse a un estado de forma instantánea. ¿Estamos a caso transmitiendo información más rápido de lo que viaja la velocidad de la luz?

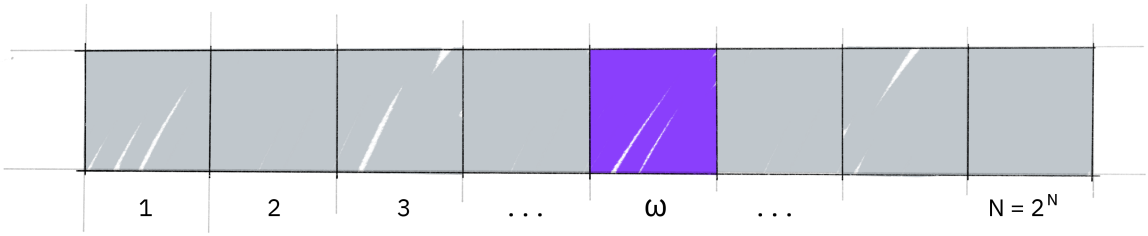
Como cabe esperar, la respuesta es no. Aunque el qubit de Bob haya colapsado a un estado, sin Bob saber la información transmitida por Alice sobre su medida, no podrá recuperar el estado original. Y, naturalmente, la medida de Alice no puede ser transmitida a Bob más rápido que la velocidad de la luz.

3.2. El algoritmo de Grover

Este algoritmo consiste en la búsqueda de un elemento particular sobre una base de datos de $n \equiv 2^N$ elementos. Esta explicación se basará en el tutorial de Qiskit y el artículo de Wikipedia sobre el algoritmo de Grover, en ingleś.

Definición del problema y oráculo

Supongamos que tenemos una función $f : \{0, 1, \ldots, N-1\} \longrightarrow \{0, 1\}$ que representa nuestra base de datos. Dado un índice x a un elemento, esta función evalúa $f(x) = 1$ si y solamente si x es el índice del elemento buscado. Dicho índice suele denominarse por ω (de *winer*).



Habitualmente accederemos a la información de f mediante un operador unitario denominado el *oráculo*, definido de la siguiente forma:

$$U_\omega|x\rangle = \begin{cases} -|x\rangle & \text{si } f(x) = 1 \\ |x\rangle & \text{si } f(x) = 0 \end{cases}$$

O, equivalentemente:

$$U_\omega|x\rangle = (-1)^{f(x)}|x\rangle$$

Supongo que disponemos de un sistema con 3 qubits y el índice buscado es $\omega = 101$. El oráculo asociado será de la forma:

$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \leftarrow \omega = 101$$

Podemos construir un oráculo genérico de la siguiente forma:

$$U_\omega = \begin{bmatrix} (-1)^{f(0)} & 0 & \cdots & 0 \\ 0 & (-1)^{f(1)} & \cdots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & (-1)^{f(2^n-1)} \end{bmatrix}$$

Amplificación de amplitudes

Nuestro algoritmo comienza con una superposición equitativa de todos los estados. Este es el estado que denominaremos $|s\rangle$:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle.$$

Si en este instante midiésemos nuestro sistema tendríamos una probabilidad de $\frac{1}{N}$ de obtener el índice ω. Aplicaremos una serie de transformaciones al sistema para aumentar la probabilidad de obtener ω al medir. Este proceso se denomina ***amplitud de amplificación***, amplificando la probabilidad de medir dicho valor.

Explicaremos el proceso a través de su interpretación geométrica. El algoritmo se describe a partir de dos reflexiones (es decir, una rotación), en un plano 2-dimensional. Los únicos estados que necesitamos considerar son el estado objetivo $|\omega\rangle$ y el estado de superposición uniforme $|s\rangle$ descrito con anterioridad. Estos dos vectores generan un plano 2-dimensional en el espacio de estados $\mathbb{C}^N$. Dado que $|\omega\rangle$ no tiene por qué ser perpendicular, podemos introducir el estado $|s'\rangle$ generado a partir de $|\omega\rangle$ y $|s\rangle$ que sí sea perpendicular a $|\omega\rangle$. Este nuevo estado puede ser generado, por ejemplo, con el algoritmo de Gram–Schmidt.

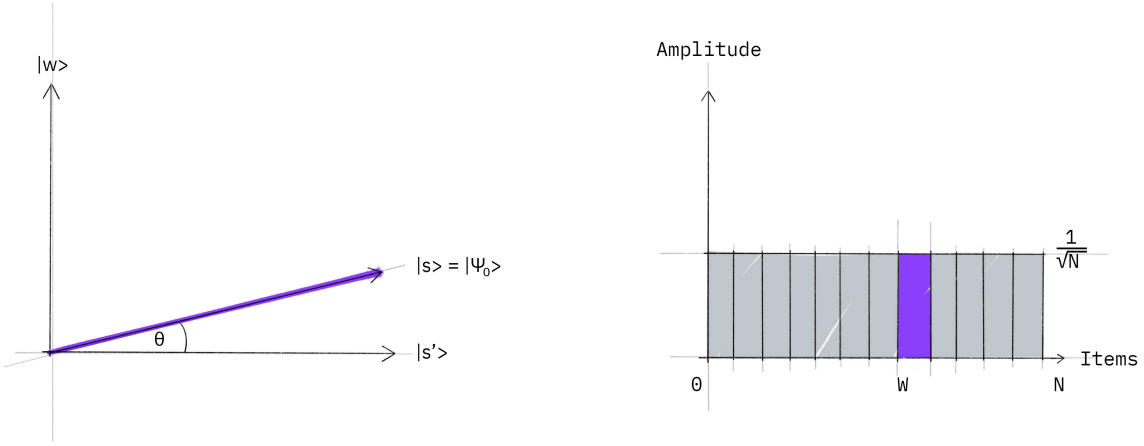


Figura 8: **Paso 1:** El proceso de amplificación de amplitud comienzo en el estado de superposición uniforme $|s\rangle$.

La imagen izquierda muestra el plano 2-dimensionale generado por $|s'\rangle$ y $|\omega\rangle$ anteriormente comentado. Esto nos permite expresar el estado inicial de la siguiente forma:

$$|s\rangle = \sin\theta|w\rangle + \cos\theta|s'\rangle; \quad \text{donde } \theta = \arcsin\langle s|w\rangle = \arcsin\frac{1}{\sqrt{N}}$$

Por otro lado, el gráfico derecho muestra las amplitudes del estado $|s\rangle$.

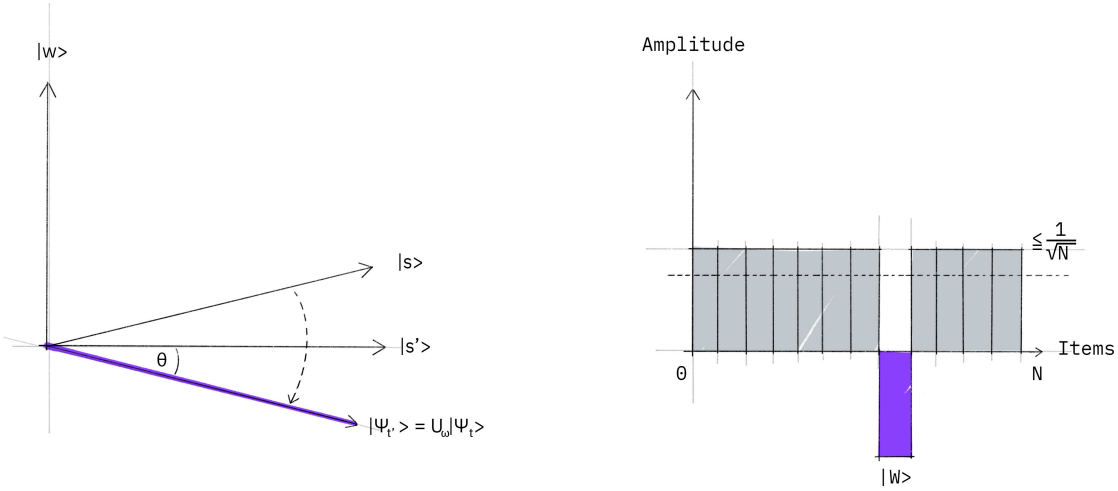


Figura 9: **Paso 2:** Aplicamos la reflexión del oráculo U_f al estado $|s\rangle$.

Geométricamente, la aplicación del oráculo U_f es una reflexión sobre el eje generado por $|s'\rangle$. Como tanto $|s\rangle$ como $|s'\rangle$ están en el plano 2-dimensional descrito, el resultado de la reflexión también estará en dicho plano. Esta reflexión se traduce en las diagrama de amplitudes en un cambio de signo para la amplitud del estado objetivo, lo que naturalmente reduce la amplitud media (representada por una línea discotínua en la gráfica.).

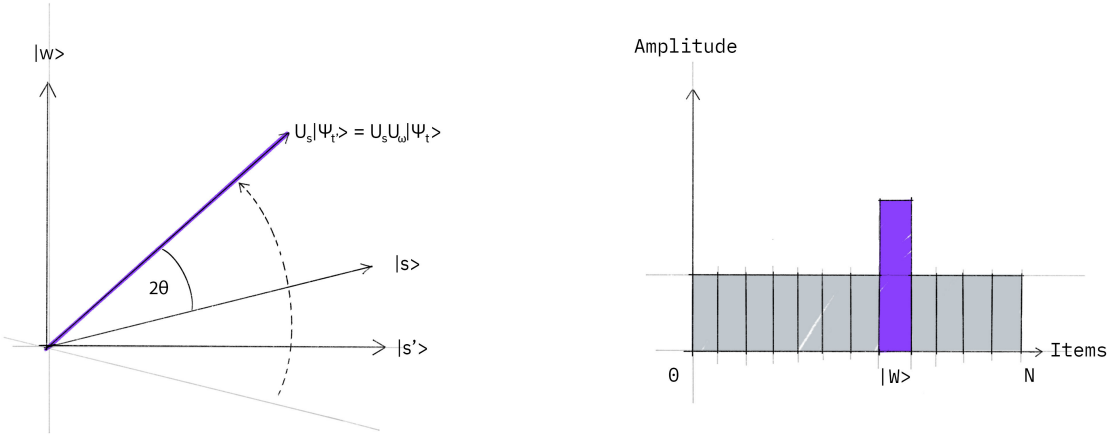


Figura 10: **Paso 3:** Aplicamos la reflexión del oráculo U_s centrada en el estado $|s\rangle$: $|U_s\rangle = 2|s\rangle\langle s| - \mathbb{1}$.

El resultado de esta nueva reflexión permanece en el plano 2-dimensional originalmente descrito. De hecho, estas dos rotaciones componen una rotación centrada en el origen. Esta tran-

formación corresponde a la aplicación del operador $U_s U_f$ a nuestro vector original $|s\rangle$. En el diagrama de amplitudes podemos entender este operador como una reflexión de la amplitud de $|\omega\rangle$ sobre la amplitud media. Puesto que la amplitud media habia sido reducido con anterioridad, este se traduce en una ampliación de la amplitud de $|\omega\rangle$ a aproximadamente tres veces su valor original, disminuyendo el resto de amplitudes.

Repitiendo este proceso t veces obtenemos el estado $|\psi_t\rangle = (U_s U_f)^t |s\rangle$. Debemos de aplicar un esta rotación durante aproximadamente $\sqrt{N}$ veces, puesto que la amplitud de $|\omega\rangle$ crece linealmente con el número de aplicaciones $\sim tN^{-\frac{1}{2}}$.

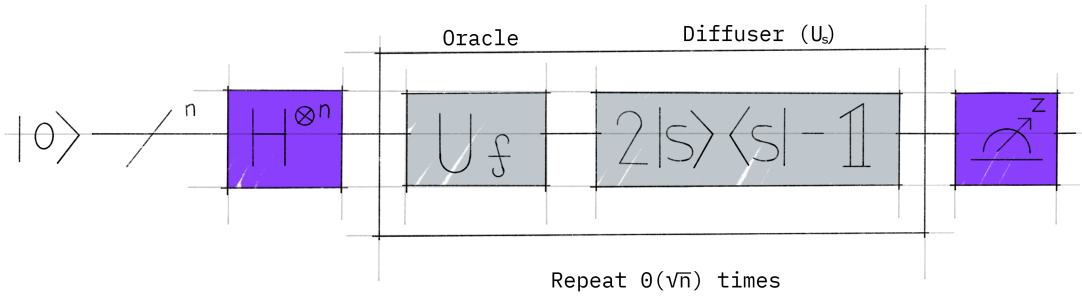


Figura 11: Circuito que implementa el algoritmo de Grover.

Como nota final, si nuestro espacio de búsqueda alberga M posible soluciones, bastará con la aplicación de dicha rotación $\sqrt{N/M}$ veces.