8. Observa que en el kernel de reducción que se presenta a continuación, para sumar N valores de un vector de números reales, la mitad de las hebras de cada bloque no hacen ningún trabajo después de participar en la carga de datos desde memoria global a un vector en memoria compartida (sdata).Modifica este kernel para eliminar esta ineficiencia y da los valores de los parámetros de configuración que permiten usar el kernel modificado para sumar N reales. ¿Habría algún costo extra en término de operaciones aritméticas necesitadas? ¿Tendría alguna limitación esta solución en términos de uso de recursos?

```
__global__ void reduceSum(float *d_V, int N)
{
        extern __shared__ float sdata[];
        int tid = threadIdx.x;
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = ((i < N) ? d_V[i] : 0.0f);
        __syncthreads();

        for (int s=blockDim.x/2; s>0; s>>=1) {
                if (tid < s) {
                        sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }
        if (tid == 0) d_V[blockIdx.x] = sdata[0];
}
```

La modificación realizada seria la siguiente:

```
__global__ void reduceSum(float *d_V, int N)
{
        extern __shared__ float sdata[];
        int tid = threadIdx.x;
```

```
        int i = blockIdx.x * (blockDim.x*2) + threadIdx.x;
        float suma = ((i < N) ? d_V[i] : 0.0f);

        if (i + blockDim.x < N)
                suma += d_V[i+blockDim.x];

        sdata[tid] = suma;

        __syncthreads();

        for (int s=blockDim.x/2; s>0; s>>=1) {
                if (tid < s) {
                        sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }
        if (tid == 0) d_V[blockIdx.x] = sdata[0];
}
```

Al añadir la suma anterior a el bucle for, estamos sumando los primeros elementos de dos bloques contiguos que se encuntran en la misma posición. De esta forma estamos ahorrando la primera pasada del bucle del ejemplo anteriro. Con esto necesitariamos en vez de N hebras N/2 hembras. Al reducir este numero de hebras en el for solo se quedaran sin ejecutar N/4 hebras en vez de las N/2 hebras que no se utilizaban para ejecutar en el ejemplo dado por el ejercicio.

El coste de operaciones necesitadas son las mismas que en el caso anterior. Puesto que la hemos quitado en cada hembra una pasada del bucle ahorrándonos una suma. Pero esa suma que ahorramos en el bucle es la utilizada fuera para obtener la primera suma. Lo único que utilizaremos de más es una multiplicación que se usara en la siguiente linea para obtener la i:

```
int i = blockIdx.x * (blockDim.x*2) + threadIdx.x;
```

Esa multiplicación sera necesaria para ir obteniendo el valor de i para cada dos bloques en vez de cada uno. Esto es necesario por lo explicado anteriormente, al sumar los elementos de dos bloques contiguos en el inicio de cada hebra.

Pero como nos quitamos una pasada del bucle al tener mensos hebras, esa multiplicación al final la estamos cambiando por la división del primer bucle. Por tanto no tendremos mas operaciones con este kernel que con el anterior.

Este nuevo kernel no tendríamos ninguna limitación en términos de recursos, puesto que al final realizamos el mismo numero de operaciones que en el anterior. Lo unico

es que las hebras tardaran mas en llegar al primer synchtreads que en el caso anterior por los nuevos cálculos.

10. Implementar un kernel para aproximar el valor de pi tal como se hace en este codigo secuencial.

```
static long num_steps = 100000;
double step;
void main()
{
        int i; double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        for (i=1;i<= num_steps;; i++){
                x = (i-0.5)*step;
                sum = sum + 4-0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Se realiza el siguiente kernel que se repetira entre cada hebra.

```
__global__ void aproximarPi( double step, double *valores, int N)
{

        int idHebra = blockIdx.x * blockDim.x + threadIdx.x;
        if(idHebra < N)
        {
                double x = (idHebra + 1 - 0.5) * step;
                valores[idHebra] = 4.0 / (1.0 + x * x);
        }
}
```

En esta funcio separamos todo lo que podria hacer cada hebra sin depender de otra. Lo unico que nos queda seria la suma. Para la suma utilizaremos simplemente la funcion de reduccion utilizada en el en el ejercicio 8.

El primer parametro sera el step que lo realizaremos fuera para no tener el mismo calculo en todas las obras. El segundo sera el vector donde se almacenaran los calculos para luego realizar la reducción con estos. El tercero sera el tamaño del vector.