RELACIÓN DE EJERCICIOS DE PROGRAMACIÓN EN CUDA C
EJERCICIOS 8 y 10 RESUELTOS

Ejercicio 8.

Propuesta de Solución (aunque se podría hacer con otros enfoques)

```
 __global__ void reduceSum(float *d_V, int N)
   {
       extern __shared__ float sdata[];
       int tid = threadIdx.x;
       int start_i= 2*blockIdx.x*blockDim.x
       int i = start_i + threadIdx.x;
       sdata[tid] = d_V[i];
       i+=blockDim.x;
       sdata[tid+blockDim.x] = ((i < N) ? d_V[i] : 0.0f);
       __syncthreads();

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

       if (tid == 0) d_V[blockIdx.x] = sdata[0];
   }
```

* **Invocación para sumar un vector A de N floats**:

```
reduceSum <<<ceil((float)N/(2*Bsize),Bsize,2*Bsize*sizeof(float) >>>  (A, N)
```

Consume el doble de espacio de memoria compartida por multiprocesador que la versión de partida, pero esto se podría compensar dividiendo por dos el tamaño de bloque ya que esta versión genera la mitad de bloques que la versión de partida.

También se podría hacer usando el mismo tamaño de bloque pero guardando, en cada posición del vector sdata, la suma de los dos valores leídos de memoria global. En este caso, se usaría el mismo espacio de memoria compartida.

Ejercicio 10.

Implementar un kernel para aproximar el valor de pi tal como se hace en este código 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;
}
```

Propuesta de Solución

Lo ideal es lanzar un grid unidimensional de bloques unidimensionales donde cada hebra calcule la suma de las aproximaciones de varios intervalos (se realiza un reparto cíclico de los intervalos entre las hebras) y después las hebras de cada bloque cooperarían para calcular la suma de las aproximaciones obtenidas. Como resultado, se obtendrían un vector de doubles con tantas celdas como bloques se han lanzado.

```
//*********************************************************
__global__ void compute_pi (double *pi_vector, int num_steps)
//*********************************************************
{
   extern __shared__ double sdata[];
   double x,sum=0.0;
   double step= 1.0/(double) num_steps;
   int idx = blockIdx.x*blockDim.x+threadIdx.x;
   int num_threads= blockDim.x*gridDim.x;
   for (int i=idx; i< num_steps; i+=num_threads) {
                       x = (i+0.5)*step;
                       sum += 4.0/(1.0+x*x);
   }
   int tid=threadIdx.x;
   sdata[tid] = sum;
   __syncthreads();
   for (int s=blockDim.x/2; s>0;s>>=1) {
                     if (tid < s)
                        sdata[tid] += sdata[tid + s];
                     __syncthreads();
   }
   if (tid == 0) pi_vector[blockIdx.x] = step*sdata[0];
 }
```

El uso en el código de este kernel podría ser el siguiente:

```
int main()
{
int num_threads=100000;

dim3 threadsPerBlock(256,1);
dim3 numBlocks( ceil ((float)(num_threads)/threadsPerBlock.x), 1);

// pi vector on CPU
double * vpi;
vpi = (double*) malloc(numBlocks.x*sizeof(double));

// pi vector  to be computed on GPU
double *vpi_d;
cudaMalloc ((void **) &vpi_d, sizeof(double)*numBlocks.x);
int smemSize = threadsPerBlock.x*sizeof(double);
// Kernel launch to compute Pi Vector
compute_pi<<<numBlocks, threadsPerBlock, smemSize>>>(vpi_d, N);
/* Copy data from device memory to host memory */
cudaMemcpy(vpi, vpi_d, numBlocks.x*sizeof(double),cudaMemcpyDeviceToHost);

// Compute pi on CPU
double pi=0.0;
for (int i=0; i<numBlocks.x;i++)
    pi+=vpi[i];
cout<<" Pi on GPU ="<<setprecision(12)<<pi<<endl;

}
```