Entrega Ejercicios 8 y 10 de CUDA

Juan Francisco Díaz Moreno

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?

La solución propuesta es la siguiente:

```
16  __global__ void reduceSum( float * d_V, int N ) {
17
18    // Vector en memoria compartida para almacenar los datos
19    extern __shared__ float sdata[];
20
21    // Cálculo de los índices para acceder al vector
22    int tid = threadIdx.x;
23    int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;
24
25    // Carga de los datos en memoria compartida (dos por hebra)
26    sdata[tid] = ( ( i < N ) ? d_V[i] + d_V[i + blockDim.x] : 0.0f );
27
28    __syncthreads();
29
30    // Reducción en memoria compartida
31    for( int s = blockDim.x/2; s > 0; s >>= 1 ) {
32      if( tid < s )
33        sdata[tid] += sdata[tid + s];
34      __syncthreads();
35    }
36
37    // Escribir el resultado en memoria principal
38    if( tid == 0 )
39      d_V[blockIdx.x] = sdata[0];
40
41  }
```

Para utilizarla, habría que lanzar la mitad de hebras que en el código inicial, pues las hebras realizan la carga inicial en memoria compartida realizando ya una de las sumas (el valor que le tocaría más uno desplazado blockDim.x posiciones, el que le tocaría a una de las hebras que nos ahorramos de esta manera).

Se produciría un gasto extra en términos de operaciones aritméticas gracias a la primera suma en la carga.

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

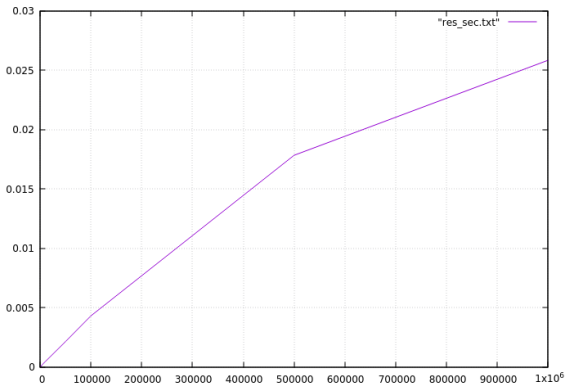
```
static long num_steps = 100000;
double step;
void main() {
    int i; double x, pi, sum = 0.0;
    step = 1.0 / (double) num_steps;
    for( i = 1; i <= num_steps; i++ ) {
        x = (i -0.5) * step;
        sum = sum + 4.0 / ( 1.0 + x * x );
    }
    pi = step * sum;
}
```

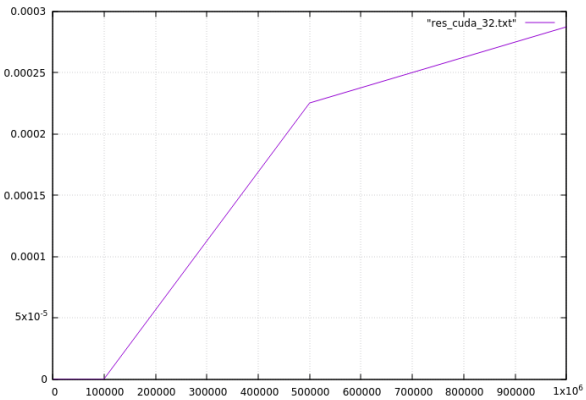
Realizar también un informe sobre los tiempos de ejecución en CPU (secuencial) y en GPU (CUDA/C++) del programa para tamaños de N elevados.

El kernel propuesto es el siguiente, tras el cual se redirigía el vector de salida (sum) al kernel del ejercicio anterior:

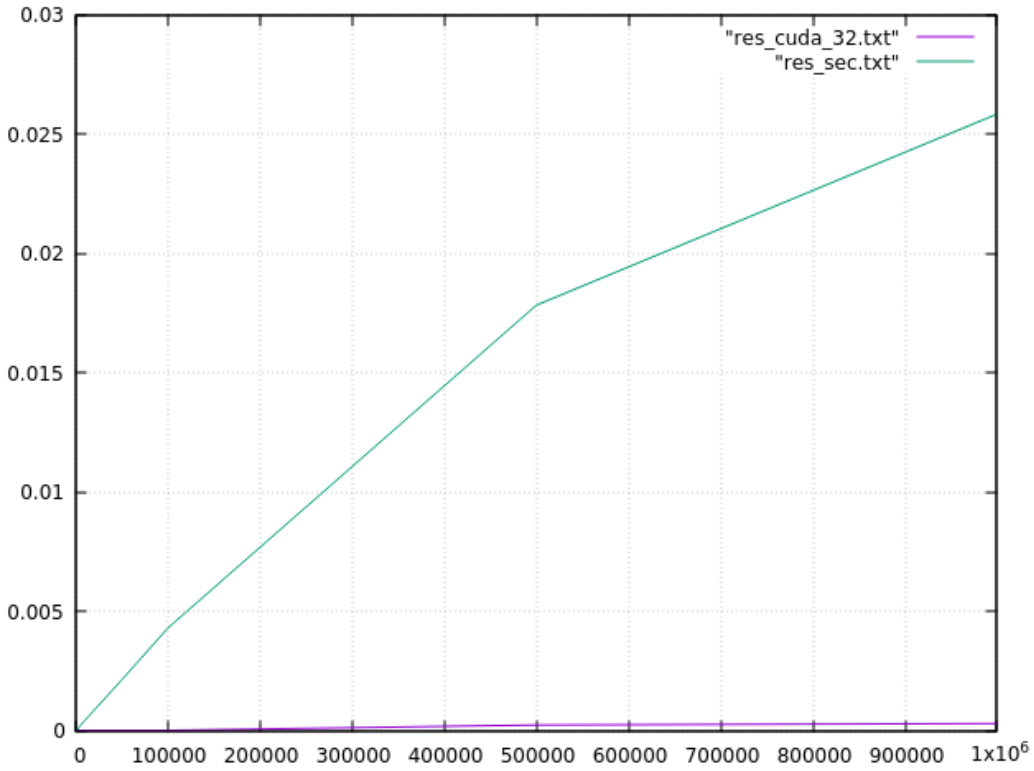
```
__global__ void calcularPI( float *sum, long num_steps, float step ) {

  double x;

  for( int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < num_steps; i += num_steps ) {
    x = ( i - 0.5 ) * step;
    sum[i] = 4.0 / ( 1.0 + x * x );
  }

}
```

Se realizó una comparación de los programas secuencial y CUDA ejecutando ambos códigos con los siguientes num_steps: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000. De esta forma se obtuvieron estas gráficas de tiempos de ejecución:





Si comparamos ambas curvas en una misma gráfica, comprobaremos que el programa CUDA se ejecuta mucho más rápido que su contraparte secuencial:



Finalmente, se probó el programa CUDA con 32, 64 y 128 hebras por bloque, obteniendo estas curvas que indican que la mejor opción para este ejercicio es 32: