Ejercicios del Tema 3 sobre Programación en CUDA C

1. Para el kernel CUDA de suma de vectores, suponiendo que el vector tiene tamaño 4000, cada hebra calcula un elemento diferente del vector de salida y la longitud del tamaño de bloque de hebras es 512 hebras. Responder a las siguientes cuestiones:

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
      {  int i = blockIdx.x * blockDim.x + threadIdx.x;
         if (i<N)      C[i] = A[i] + B[i];
      }
```

a) ¿Cuántas hebras habrá en la Grid asociada al kernel?

b) ¿Cuántos bloques de hebras se procesarán en total?

c) ¿Cuántas hebras del kernel no harán trabajo útil?

d) Escribir la parte del código en la que se lanzaría el kernel, si quisiéramos sumar vectores de 3000 enteros y se usaran bloques de 128 hebras. ¿Cuál sería el número de hebras de la grid asociada?

2. Modificar el kernel del ejercicio 1 para que cada hebra CUDA calcule dos elementos adyacentes del vector resultado de la suma en lugar de un único elemento.

3. Supongamos que necesitamos escribir un kernel que opera sobre una imagen de tamaño 400 x 900 pixels. Deseamos asignar una hebra para los cálculos asociados a cada pixel. También queremos usar bloques de hebras cuadrados y el mayor número posible de hebras por bloque (asumiendo capacidad de cómputo 3.0).
a) Indicar cómo habría que seleccionar las dimensiones de la grid y de los bloques del kernel para dicho propósito.
b) Indicar también cuántas hebras ociosas esperarías tener.

4. Un programador CUDA dice que si lanza un kernel con solo 32 hebras en cada bloque, no necesitará usar la instrucción __syncthreads() cuando una sincronización de barrera es necesaria a nivel de bloque de hebras. Explicar si crees que esto es una buena idea.

5. ¿Qué tipo de comportamiento de ejecución incorrecto podría darse si uno olvida la instrucción __syncthreads() en el kernel de advección lineal basado en memoria compartida?

6. Supongamos que un kernel se lanza con 1000 bloques de hebras, cada uno con 512 hebras.

a) Si una variable se declara como local al kernel, ¿Cuántas versiones de dicha variable se crearán a lo largo de la ejecución del kernel?

b) Si una variable se declara como una variable en memoria compartida, ¿Cuántas versiones de dicha variable se crearán a lo largo de la ejecución del kernel?

7. Se dispone del siguiente kernel para calcular la suma de matrices cuadradas N x N de elementos de tipo float, donde cada hebra tiene asignado el cálculo de una celda de la matriz resultado C:

```
__global__ void MatAdd (float *A, float *B, float * C, int N)
       {
        int j = blockIdx.x * blockDim.x + threadIdx.x;
        int i = blockIdx.y * blockDim.y + threadIdx.y;
        int index=i*N+j;
        if (i < N && j < N)
               C[index] = A[index] + B[index];
        }
```

a) Modifica este kernel para que cada hebra calcule una fila de la matriz resultado y escribe los parámetros de la configuración de ejecución para sumar dos matrices N x N .

b) Modifica este kernel para que cada hebra calcule una columna de la matriz resultado y escribe los parámetros de la configuración de ejecución para sumar dos matrices N x N .

c) Analiza los pros y los contras que presenta cada uno de los tres kernels mostrados.

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];
  }
```

9. Dado un vector A de N valores reales almacenado en memoria del dispositivo GPU, diseñar un kernel CUDA para invertir el orden del vector A en otro vector en GPU B.

a) Describir un kernel basado en un único bloque de 256 hebras asumiendo que N=256.

b) Describir un kernel para invertir vectores con N siendo un múltiplo de 256 basado solo en accesos a memoria global. ¿Qué problemas de eficiencia presenta este kernel?

c) Describir un kernel para cualquier N múltiplo de 256 pero que resuelva los problemas del apartado b) usando memoria compartida.

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;
```

11. El siguiente kernel se ha ejecutado sobre una gran matriz, que es dividida en submatrices contiguas. Para manipular las submatrices, un nuevo programador CUDA ha escrito el siguiente kernel, que transpondrá cada submatriz en la matriz. Las submatrices son de tamaño BLOCK_WIDTH x BLOCK_WIDTH, y cada una de las dimensiones de la matriz A es múltiplo de BLOCK_WIDTH . Se muestran abajo la invocación del kernel y el código del mismo donde BLOCK_WIDTH se conoce en tiempo de compilación y está entre 1 y 20.

```
dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH);
dim3 gridDim(A_width/blockDim.x, A_height/blockDim.y);

BlockTranspose <<< gridDim, blockDim >>> (A, A_width, A_height);


__global__ void BlockTranspose(float * A_elements, int A_width, int A_height)
{
     __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];

     int baseIdx = blockIdx.x * BLOCK_WIDTH + threadIdx.x;
     baseIdx   + = (blockIdx.y * BLOCK_WIDTH + threadIdx.y)  * A_width;
     blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];
     A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}
```

a) ¿Para qué valores de BLOCK_WIDTH este kernel funcionará correctamente al ejecutarse sobre una GPU?

b) Si el código no se ejecutara correctamente para todos los valores de BLOCK_WIDTH, sugiere cómo arreglarlo para hacer que trabaje para todos los valores de BLOCK_WIDTH.