Resolución del Ejercicio 7 de la relación de Ejercicios de Programación de GPUs

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 .

La adaptación del kernel para que cada hebra calcule una fila de la matriz resultado es inmediata si se consideran bloques unidimensionales donde a cada hebra se le asigna un índice de fila:

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

La invocación del kernel podría ser como se describe a continuación:

```
int numBlocks = ceil(float(N)/blockSize);
MatAdd<<<numBlocks, blockSize>>>(a_d, b_d, c_d, 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 .

De la misma forma que se ha hecho anteriormente, también se puede adaptar el kernel para que cada hebra calcule una columna de la matriz:

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

La invocación se haría igual que en el caso a).

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

En teoría, el kernel que calcula un elemento por hebra es bastante eficiente ya que permite trabajar con un número de hebras mucho mayor que los dos casos anteriores (incluso con N bajo se lanzan N*N hebras). No obstante, el acceso a datos no es completamente coalescente en este kernel.

El peor kernel es el que calcula una fila por hebra, ya que el acceso a los datos dentro de cada warp hace que cada hebra acceda a un elemento separado N posiciones en memoria con respecto elemento accedido por la siguiente hebra del warp. Esto hace que cada acceso a datos en memoria global que hace un warp de hebras requiera de un número de transaccciones muy elevado.

El kernel que calcula una columna por hebra permite un acceso a datos en memoria global de forma altamente coalescente (en término medio) pero está limitado si se trabaja con matrices de dimensión no muy elevada. En cambio, si se trabaja con matrices de dimensión muy grande podría conseguir tiempos mejores que el primer kernel al permitir un buen aprovechamiento de todos los multiprocesadores de la GPU y beneficiándose de un acceso muy eficiente a memoria global.