ARQUITECTURA DE SISTEMAS PARALELOS
Coprocesadores gráficos GPU

Problemas Tema 4:

4.1.- Diferencie las características de los siguientes tipos de memoria presentes en una GPU:

Registros- Memoria compartida- memoria global- memoria de textura – memoria constante.

- Indique que tipo de datos es más adecuado utilizar en cada caso.
- De tódos ellos cuál es el que está vinculado al número máximo de threads que se pueden utilizar en cada bloque.

4.2- La arquitectura de una GPU se caracteriza por:

- Número máximo de Threads por bloque es 512.
- El tamaño de warp es 32.
- El número de registros por multiprocesador SM es 8192.
- El número máximo de bloques que pueden correr simultáneamente en un multiprocesador SM es de 8
- El número máximo de warp que pueden correr simultáneamente en un multiprocesador SM es de 24.

Para una multiplicación de matrices que distribución de threads por bloque (tamaño de bloque) de las siguientes es la más adecuada

a) 8x8
b) 16x16
c) 32x32

Justifique la respuesta

4.3- En la arquitectura anterior:

¿Qué se puede concluir en términos de reparto de threads, si en una aplicación está ejecutando simultáneamente 24 Warps en uno de los multiprocesadores SM?

4.4- En la arquitectura anterior, suponga que :

- Para pasar a ejecución (dispatch) todos los threads de un warp se necesitan 4 ciclos de reloj.
- Un kernel realiza un acceso a memoria global se produce cada 4 instrucciones.
- Cada acceso a memoria global supone una latencia de 200 ciclos.

Estime el número de Warps que debe estar ejecutando el sistema para ocultar las penalizaciones de acceso a memoria.

4.5- ¿Qué tipo de comportamiento incorrecto puede ocurrir si olvidamos utilizar la instrucción __syncthreads() en el kernel que se muestra a continuación?. Existen dos llamadas a la función __syncthreads() justifica cada una de ellas.

```
__global__
void matrix_mul_kernel(float* Md, float* Nd, float* Pd, const int cWidth)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx_ = blockIdx.x;
  int by_ = blockIdx.y;
  int tx_ = threadIdx.x;
  int ty_ = threadIdx.y;

  int row_ = by_ * TILE_WIDTH + ty;
  int col_ = bx_ * TILE_WIDTH + tx;

  float p_value_ = 0.0f;

  for (int m = 0; m < cWidth / TILE_WIDTH; ++m)
  {
    Mds[ty_][tx_] = Md[row_ * cWidth + (m * TILE_WIDTH + tx_)];
    Nds[ty_][tx_] = Nd[(m * TILE_WIDTH + ty_) * cWidth + col_];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      p_value_ += Mds[ty_][k] * Nds[k][tx_];

    __syncthreads();
  }

  Pd[row_ * cWidth + col_] = p_value_;
}
```

4.6. Suponga que un SM (Stream Multiprocesor) tiene las siguientes características.

SM Resoruces:

- Maximun number or warps per SM = 64
- Maximun number or blocks per SM =32
- Register usage = 256 KB
- Available Shared memory = 64 KB.

Se quiere ejecutar un kernel, con el número de bloques nBlk y cada bloque con nThr Treads
.
Kernel <<< nBlk,bThr >>> (…)

Teniendo en cuenta que en el código del kernel se utiliza memoria compartida como se indica en el fragmento de código:

```
__global__ Kernel (…) {
        __shared__ int A [1024];        // tamaño de un int es 4 bytes
        int x =4;
        index= blockIdx.x * blockDim.x + threadIdx.x
        for ( a = 0, a <MAX, a++) {
           m[a] = 2* A[index+…] * C;
          …
        }
…
}
```

En estas condiciones ¿cuál es el número máximo de Bloques que pueden ejecutarse al lanzar este kernel en el SM?

Al duplicar el valor de MAX (índice del bloque) se detecta al compilar el programa que el número de registros usados por bloque pasa de ser 8K a 32K. ¿Cómo se modifica la situación anterior?

4.7. Suponga que tiene un vector de 2^{27} floats (2^{27} = 134.217.728 = 1024 *131.072) y que se utiliza una GPU para calcular la suma de los elementos. Los siguientes códigos son intentos incorrectos o ineficientes. Explique cómo funcionan identificando las causas de los problemas, dando una breve explicación indicando lo que esté mal, y/o lo que genere ineficiencias.

a) Se crea un kernel para ejecutar con 134.217.728 threads.

```
0 __global__ void sum1(float *x, int n) {
1       // assume n is a power of 2
2       int myId = blockDim.x * blockIdx.x + threadIdx.x;
3       if(myId < n) {
4               for(int stride = 2; stride < n; stride *= 2) {
5               __syncthreads();
6               if ((myId % stride) == 0)
7                       x[myId] = x[myId] + x[myId + stride]
8               }
9       }
10      // the result is in x[0]
11 }
```

El kernel se lanza con: sum1<<<1, 134217728>> (x, 134217728)

b) Se crea el mismo kernel para ejecutar con 134.217.728 threads pero el kernel se lanza con: sum1<<<131072, 1024>> (x, 134217728).

c) Se crea un kernel para lanzar un bloque con 1024 threads y cada thread se encarga de la suma de n/1024 elementos antes de realizar el proceso de reducción para obtener la suma.

```
0 __global__ void sum3(float *x, int n, int m){
1       // assume n and m are powers of 2
2       int myId = blockDim.x * blockIdx.x + threadIdx.x;
3       f(myId < n) {
4       // compute the sum for my part of the array
5       float sum = 0.0;
6        for(int i = 0; i < m; i++)
7               sum += x[myId + i];
8       // reduce
9       for(int stride = 2; stride < blockDim.x; stride *= 2) {
10      __syncthreads();
11              if((myId % stride) == 0)
12              x[myId] = x[myId] + x[myId + stride]
13              }
14      }
15      // the result is in x[0]
16 }
```

y el kernel se lanza con: sum3<<<1, 1024>>>(x, 134217728, 131072)

d) Minimizando el número de líneas a cambiar en el kernel sum3 del apartado c) resuelva los problemas encontrados

4.8. Un programador inexperto de CUDA está tratando de optimizar su primer kernel de GPU para el rendimiento. Quiere encontrar la mejor configuración de ejecución (es decir, el tamaño de grid, el tamaño de bloque, y el número de threads por bloque). A medida que asigna un thread por elemento de entrada, calcula el tamaño de grid (es decir, el número total de bloques) de la siguiente manera. Para N elementos de entrada, el tamaño de grid es [N/block_size] , donde block_size es el número de hilos por bloque. La parte que debe optimizar, será descubrir cuál es el tamaño de bloque que produce el mejor rendimiento, intentando 5 tamaños de bloque diferentes posibles para su GPU (64, 128, 256, 512 y 1024 Threads).

Una recomendación general para la optimización del kernel es maximizar la ocupación de todos los Stream Multiprocesors(SM) de la GPU. La ocupación se define como la proporción de Threads activos respecto al número máximo posible de threads por SM.

Para calcular la ocupación, es necesario tener en cuenta los recursos disponibles. Se sabe que en cada SM de su GPU:

- la memoria compartida es de 16 KB.
- El número total de registros de 4 bytes es 16384.

En una primera versión del código del kernel, cada thread utiliza 2 elementos de 4 bytes en la memoria compartida. Además, cada bloque, independientemente de su tamaño, necesita 16 elementos adicionales de 4 bytes en la memoria compartida para la comunicación entre hilos.
Para determinar el uso de registros, la cantidad de registros que necesita cada Thread se investiga mediante el uso de una bandera del compilador y se obtiene que cada hilo en la primera versión del kernel usa 9 registros.

a) Suponiendo que el número de bloques está limitado por la memoria compartida cuál de las opciones anteriores (64, 128, 256, 512 y 1024 Threads/Bloque) es la más adecuada.

Otras limitaciones de la GPU pueden modificar la elección anterior. Restricciones de hardware de cada SM.

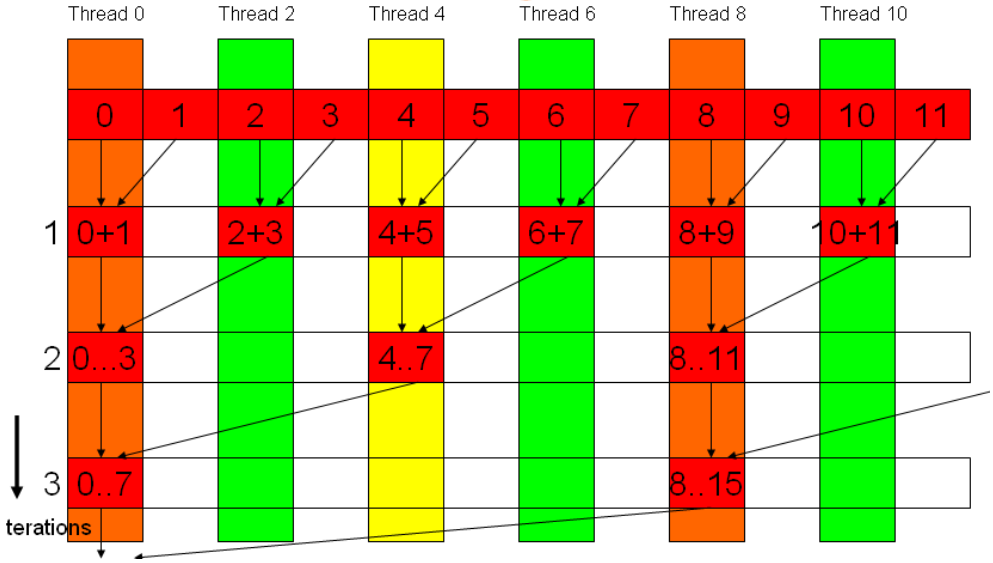
- El número máximo de bloques por SM es 8.
- El número máximo de hilos por SM es 2048.

b) Con estas nuevas restricciones, ¿cuál de las opciones anteriores (64, 128, 256, 512 y 1024 Threads/Bloque) es la más adecuada?

c) Considerando las restricciones del apartado b), cuantos registros se utilizan en cada una de las opciones (64, 128, 256, 512 y 1024 Threads/Bloque) y cual esta más cerca de alcanza la limitación por registros?

d) El rendimiento obtenido por la primera versión del kernel no cumple con la aceleración necesarias. Por lo tanto, el programador escribe una segunda versión del kernel que reduce la cantidad de instrucciones a expensas de usar un registro más por hilo. ¿Cuál sería la ocupación más alta para el segundo kernel? ¿Para qué tamaño de bloque se consigue?

4.9. Se necesita realizar una reducción en un sistema que utiliza una GPU para almacenar la suma de todos los elementos de un vector en el elemento 0 del mismo vector.



Un programador realiza una primera versión del programa que realiza la reducción partiendo de las siguientes condiciones:

- El vector original se encuentra almacenado en la memoria global de la GPU.
- La memoria compartida se utiliza para guardar la suma parcial.
- Cada iteración realiza una suma parcial de acuerdo a la figura.
- La solución final se almacena en el elemento 0.

La parte del código de interés donde se asume el vector ya cargado es la siguiente:

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
       stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
       partialSum[t] += partialSum[t+stride];
}
```

Para una ejecución sobre un vector muy grande (>10.000), se paraleliza con un tamaño de bloque de 512, y para ello si es necesario se añaden elementos adicionales al vector de valor cero.

Se pide:

a) Explicar el funcionamiento indicando la mejora en rendimiento respecto a una versión no paralelizada.
b) Detalle como se comporta el sistema en términos de eficiencia.
c) Indique, justificando la respuesta, cuantos warps, están activos por iteración.
d) Introduzca alguna mejora en el código anterior que mejore el funcionamiento de la aplicación. Represente con un esquema similar a la figura la ejecución del nuevo código y justifique la razón por la que se espera mejorar.

Detalle como cambia, si es el caso la ejecución de threads, warps y el acceso a memoria.

4.10. Para el siguiente código escrito en CUDA. ¿ Cómo se podría reescribir para mejora las transferencias a memoria global y, si procede, a memoria global. Explique la respuesta.

NOTA: c is stored in row-major order, so c[i][j] is adjacent to c[i][j+1].

```
N = 512;
NUMBLOCKS = 512/64;

float a[512], b[512], c[512][512];

 __global compute(float a, float *b, float *c) {
 int tx = threadIdx.x;
 int bx = blockIdx.x;
for (j = bx*64; j< bx+64; j++)
a[tx] = a[tx] - c[tx][j] * b[j];
 }
```

4.11. Para el siguiente código escrito en CUDA, Describa cómo se podría modificar para mejora la divergencia en los saltos. Explique la respuesta.

NOTA: The functions *starting_kernel* and *default_kernel* compute b from a in different ways

```
main() {
float h_a[1024], h_b[1024];

… /* assume appropriate cudaMalloc called to create d_a and d_b, and d_a is */
 /* initialized from h_a using appropriate call to cudaMemcpy */

dim3 dimblock(256);
dim3 dimgrid(4);
compute<<<dimgrid, dimblock,0>>>(d_a,d_b);
/* assume d_b is copied back from the device using call to cudaMemcpy */
}

 __global__ compute (float *a, float *b)
{ float a[4][256], b[4][256];
int tx = threadIdx.x;
 if (tx % 16 == 0)
     (void) starting_kernel (a[bx][tx], b[bx][tx]);
else /* (tx % 16 > 0) */
  (void) default_kernel (a[bx][tx], b[bx][tx]);
}
```

4.12. El siguiente código realiza la correlación de una región de una imagen con un patrón. Describa cómo se podría distribuir la imagen y el patrón para que ajuste en una memoria global de 128 MB para la imagen y 16 KB de memoria compartida para el patrón (template) . Explique la respuesta.

TEMPLATE_NROWS = TEMPLATE_NCOLS = 64;
IMAGE_NROWS = IMAGE_NCOLS = 5192;
int image[IMAGE_NROWS][IMAGE_NCOLS], th[IMAGE_NROWS][IMAGE_NCOLS];
int template[TEMPLATE_NROWS][TEMPLATE_NCOLS];

```
for(m = 0; m < IMAGE_NROWS - TEMPLATE_NROWS + 1; m++)
      for(n = 0; n < IMAGE_NCOLS - TEMPLATE_NCOLS + 1; n++)
            for(i=0; i < TEMPLATE_NROWS; i++)
                  for(j=0; j < TEMPLATE_NCOLS; j++)
                        if(abs(image[i+m][j+n] – template[i][j]) < threshold)
                              th[m][n]+= image[i+m][j+n]
```

4.13. Explique brevemente como soluciona una GPU las situaciones enumerada a continuación y justifica la necesidad de que el rendimiento se consigue por la ejecución de miles de threads repartidos entre los SMs.

- Ocultamiento de las penalizaciones por riesgos de datos.
- Ocultamiento de las penalizaciones por riesgos de control.
- Ocultamiento de latencias en el acceso a memoria global.

4.14.

If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index:

(A) i=threadIdx.x + threadIdx.y;
(B) i=blockIdx.x + threadIdx.x;
(C) i=blockIdx.x*blockDim.x + threadIdx.x;
(D) i=blockIdx.x * threadIdx.x;

4.15.

Assume that we want to use each thread to calculate two (adjacent) elements of a vector addition. What would be the expression for mapping the thread/block indices to I, the data index of the first element to be processed by a thread?

(A) i=blockIdx.x*blockDim.x + threadIdx.x +2;
(B) i=blockIdx.x*threadIdx.x*2
(C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2
(D) i=blockIdx.x*blockDim.x*2 + threadIdx.x

4.16.

We want to use each thread to calculate two elements of a vector addition. Each thread block process 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

(A) i=blockIdx.x*blockDim.x + threadIdx.x +2;
(B) i=blockIdx.x*threadIdx.x*2
(C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2
(D) i=blockIdx.x*blockDim.x*2 + threadIdx.x