Procesamiento de Grandes Volumenes de Datos
Coprocesadores gráficos GPU

Planificación de Threads:

3.1.- 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

3.2.- 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?

3.3.- 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.

3.4.- 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?

3.5.- 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?

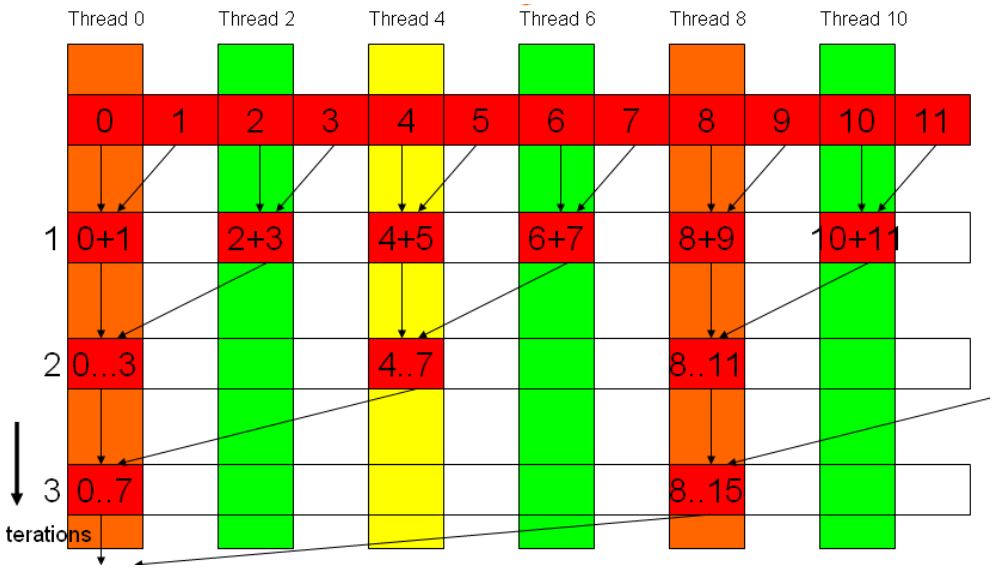
d) El rendimiento obtenido por la primera versión del kernel no cumple con la aceleración necesarias. Por lo tanto, el programador escribe una segunda versión del kernel que reduce la cantidad de instrucciones a expensas de usar un registro más por hilo. ¿Cuál sería la ocupación más alta para el segundo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

3.6.- ¿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_;
}
```

3.7.- 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.

3.8.- Analice el acceso a memoria del siguiente programa.

```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

Suponga que el kernel se lanza con bloques de 16 x 2 threads y considere para el análisis de acceso asuma que n = 1000.
¿Qué efecto tendría modificar el acceso a memoria intercambiando los papeles de i y j?

```
    for (int k = 0; k < n; ++k) {
        float x = d[n*j + k];
        float y = d[n*k + i];
        float z = x + y;
        v = min(v, z);
    }
```

3.9.- Considere el siguiente fragmento de un programa CUDA

```
__global_void my_kernel (int *a, int * b)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x ;   /*note that blockDim.x = 2 */
        a[idx+1] = threadIdx.x  ;
        b[idx] = blockIdx.x  ;
}
void main(){
   …
my_kernel<<< 4,2 >>> (a , b)   /* four blocks, each containing 2 threads */
…
}
```

Suponga que el vector a[] y b [] están reservados en la memoria global y están inicializado a -1.

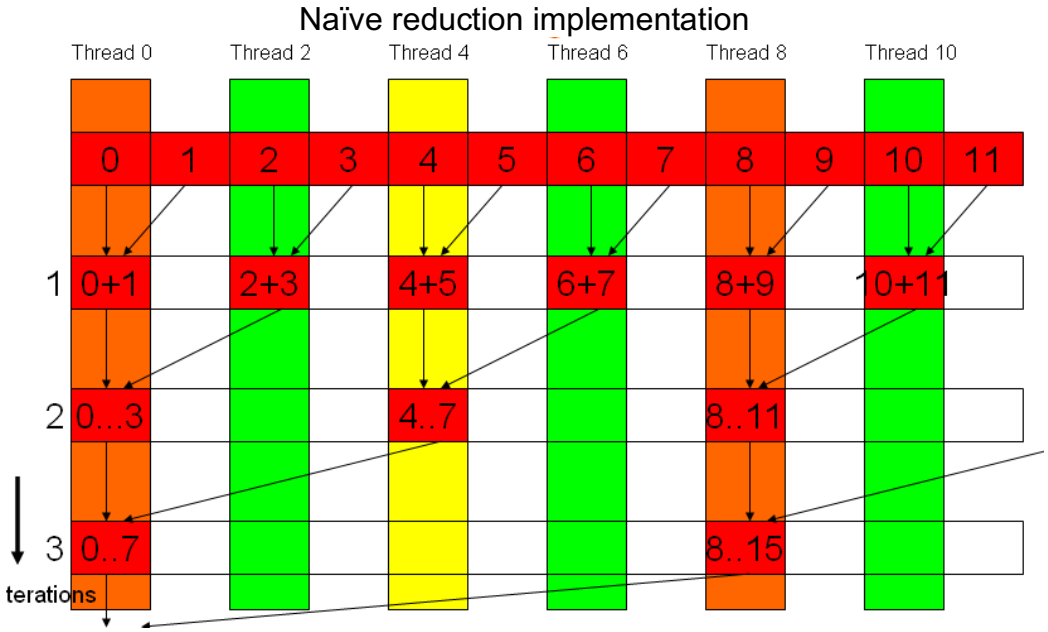
¿ Cuál será los valores almacenados después de la ejecución del kernel?

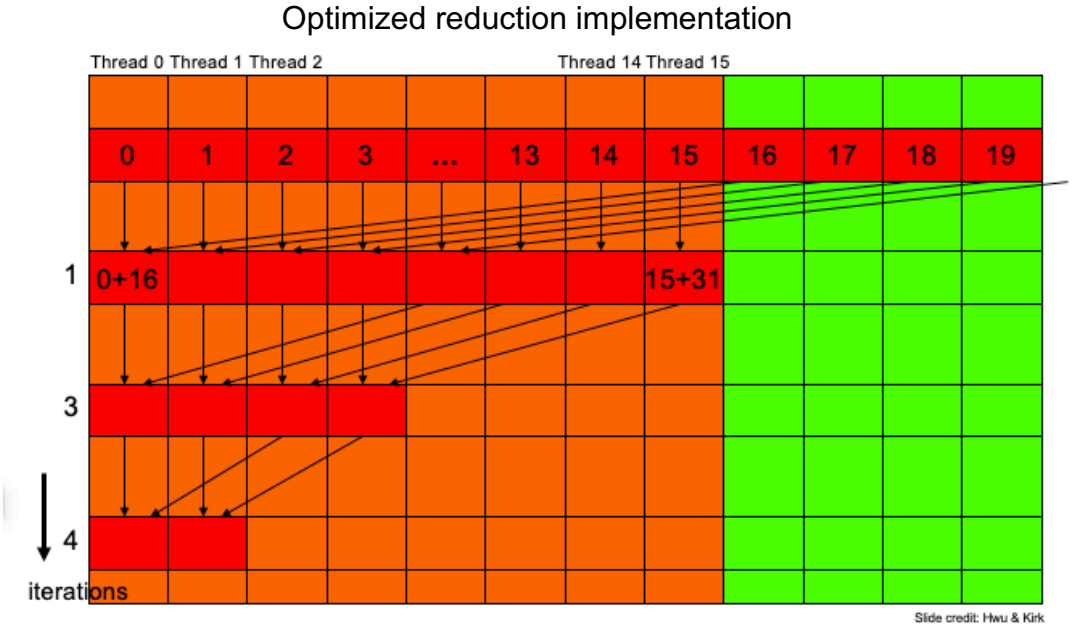
3.10.- En la siguiente llamada de un kernel en Cuda

Kernel<<< dim3 (8,4,2), dim3(16,16) >>> (…)

- ¿Cuántos bloques se están lanzando?
- ¿Cuántos threads por bloque?
- ¿Cuántos threads en total en el Stream Multiprocessor(SM)?

3.11.- Para dos kernels de reducción que se implementan par GPU según las figuras y suponiendo que operan con 256 elementos en memoria global utilizando un bloque de 256 threads, conteste justificadamente las siguientes preguntas:





a. ¿Qué eficiencia y aceleración se espera conseguir para cada uno de los kernels GPU comparando con la ejecución de la suma serie de n=256 elementos en una CPU con una operación de suma de dos operandos?
b. Para el kernel denominado "naïve reduction", ¿Cuantos pasos (iteraciones en la reducción) se necesitan?
c. Explique que es la divergencia de threads e indique para la implementación "naïve reduction" cuántos pasos tienen divergencia indicando la razón.
d. ¿Para la implementación denominada "optimized reduction" indique que pasos tiene divergencia de threads y cuáles no?
e. El kernel denominado Implementación optimizada ¿se beneficiaría del uso de memoria compartida? Detalle como se realizaría e indique por qué o por qué no.

3.12.- Para el siguiente kernel de suma de dos vectores y el código correspondiente que se utiliza para su ejecución, conteste justificadamente las siguientes preguntas:

```
1  __global__ void vecAddKernel (float* A, float* B, float* C, int n)
2  {
3    int i = threadIdx.x + blockDim.x * blockIdx.x * 2;
4
5    if (i < n) { C_d[i] = A_d[i] + B_d[i]; }
6    i += blockDim.x;
7    if (i < n) { C_d[i] = A_d[i] + B_d[i]; }
8  }
9
10 int vectAdd (float* A, float* B, float* C, int n)
11 {
12   // Parameter "n" is the length of arrays A, B, and C.
13   int size = n * sizeof (float);
14   cudaMalloc ((void **)&A_d, size);
15   cudaMalloc ((void **)&B_d, size);
16   cudaMalloc ((void **)&C_d, size);
17   cudaMemcpy (A_d, A, size, cudaMemcpyHostToDevice);
18   cudaMemcpy (B_d, B, size, cudaMemcpyHostToDevice);
19
20   vecAddKernel<<<ceil (n / 2048.0), 1024>>> (A_d, B_d, C_d, n);
21   cudaMemcpy (C, C_d, size, cudaMemcpyDeviceToHost);
22 }
```

a. Si el tamaño n de los vectores A, B y C es de 50,000 elementos, cada uno. ¿Cuántos bloques de threads se generan?

b. Si el tamaño n de los vectores A, B y C es de 50,000 elementos, cada uno. ¿Cuántos warps hay en cada bloque de threads?

c. Si el tamaño n de los vectores A, B y C es de 50,000 elementos, cada uno. ¿Cuántos threads en total se generan en el grid lanzado en la línea 20?

d. Si el tamaño n de los vectores A, B y C es de 50,000 elementos, cada uno. Indique sobre que elementos actúa el primer y último thread del primer, segundo y último bloque.

e. Si el tamaño n de los vectores A, B y C es de 50,000 elementos, cada uno. ¿Hay divergencia de threads en la ejecución del kernel? Explique cuando se produce y cuando no, identificando el número de bloques y de warps con divergencia. Justifique identificando las líneas de código que hayan generado la divergencia para cada caso.

f. Indique una desventaja en términos de rendimiento de este kernel de suma de vectores, que calcula dos elementos del vector resultado por thread, en comparación con un kernel que solo calcule un elemento del vector resultado por thread.