Universidad Autónoma de Madrid

Escuela Politécnica Superior

Tema 3: Paralelismo de datos a gran escala con GPU

Sesión de problemas

Lista de ejercicios que quiero hacer

- Ejercicio 3.1 (hecho en teoría)
- Ejercicio 3.3
- Ejercicio 3.4
- Ejercicio 3.5
- Ejercicio 3.6
- Ejercicio 3.8
- Ejercicio 3.11

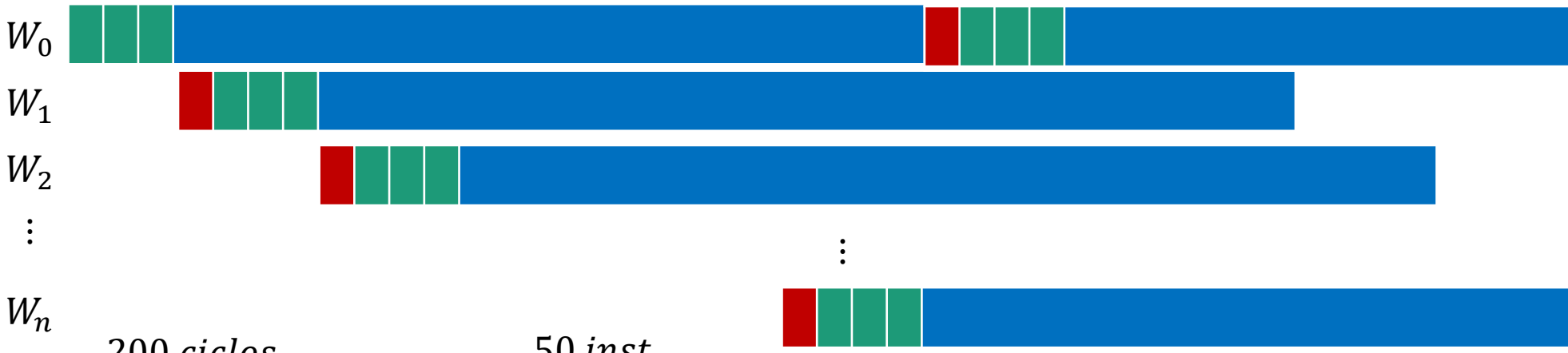
3.3.- En la arquitectura anterior, suponga que :

- 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.

La solución es para la arquitectura G80. Datos adicionales:

- 4 ciclos por instrucción para todas las instrucciones (salvo memoria).
- Solo puede haber un warp a la vez en ejecución.



$$\frac{200\ ciclos}{4\ inst/ciclo} = 50\ inst \qquad \frac{50\ inst}{4\ inst/warp} = 12.5\ warps \Rightarrow 13\ warps$$

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.

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;
           …
        }
…
}
```

1024*4B=4KB por bloque

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

¿Qué recurso nos limita?

La memoria compartida

$$\frac{64KB}{4KB/bloque} = 16\ bloques$$

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.

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?

¿Qué recurso nos limita?

Los registros y la memoria compartida

$$\frac{256KB}{32KB/bloque} = 8\ bloques$$

$$\frac{64KB}{4KB/bloque} = 16\ bloques$$

Elegimos el más restrictivo: 8 bloques

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.

• 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.

M	Shared/block (8*M+64)	Máx. Blocks (16KB/Shared/block)
64	576	28
128	1088	15
256	2112	7
512	4160	3
1024	8256	1

• 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.

M	Máx. Blocks (16KB/Shared/block)	Total threads
64	28	1792
128	15	1920
256	7	1792
512	3	1536
1024	1	1024

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?

M	Máx. Blocks (16KB/Shared/block)	Total threads
64	8	512
128	8	1024
256	7	1792
512	3	1536
1024	1	1024

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.

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?

M	Máx. Blocks (16KB/Shared/block)	Total threads	Total registers (Total threads*9)
64	8	512	4608
128	8	1024	9216
256	7	1792	16128
512	3	1536	13824
1024	1	1024	9216

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.

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?

M	Máx. Blocks (16KB/Shared/block)	Total threads	Total registers (Total threads*10)
64	8	512	5120
128	8	1024	10240
256	~~7~~ 6	~~1792~~ 1536	15360 ~~17920~~
512	3	1536	15360
1024	1	1024	10240

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.

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

```
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();
}
```

- Barrier 1: esperar a escribir la memoria compartida.
- Barrier 2: esperar a que todos terminen la iteración del bucle y no pisar Mds antes de tiempo.

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.

Hay que analizar los dos accesos 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);
```

d[0]	d[1000]	d[2000]	d[3000]	d[4000]	d[5000]	d[6000]	d[7000]	d[8000]	d[9000]	d[10000]	d[11000]	d[12000]	d[13000]	d[14000]	d[15000]
d[0]	d[1000]	d[2000]	d[3000]	d[4000]	d[5000]	d[6000]	d[7000]	d[8000]	d[9000]	d[10000]	d[11000]	d[12000]	d[13000]	d[14000]	d[15000]
d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]
d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]	d[1]

El primer acceso a memoria es aleatorio y no secuencial.

El segundo acceso es constante.

¿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);
}
```

d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]	d[0]
d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]	d[1000]

d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]	d[13]	d[14]	d[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]	d[13]	d[14]	d[15]

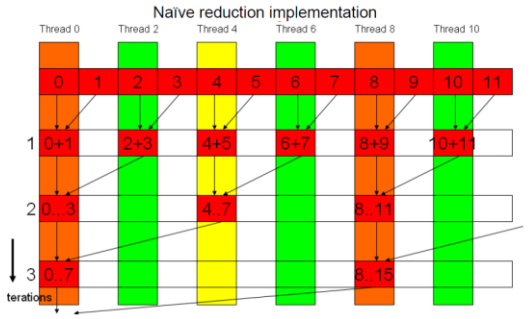
El primer acceso a memoria es constante.

El segundo acceso es secuencial.

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:

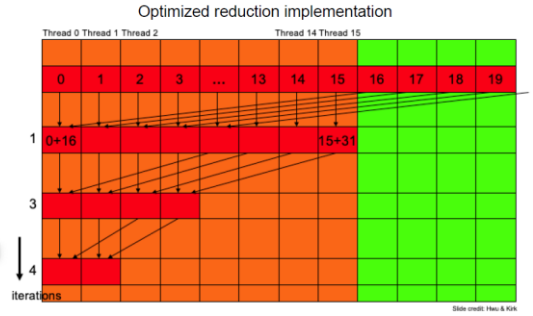
a. ¿Qué eficiencia y aceleración se espera conseguir para cada uno de los kernels GPU comparando con la ejecución de la suma serie de n=256 elementos en una CPU con una operación de suma de dos operandos?

$$S(256) = \frac{T(1)}{T(256)} = \frac{255\ u.t.}{8\ u.t} \qquad \mathrm{E}(256) = \frac{S(256)}{256} = \frac{255}{256} \cdot \frac{1}{8}$$



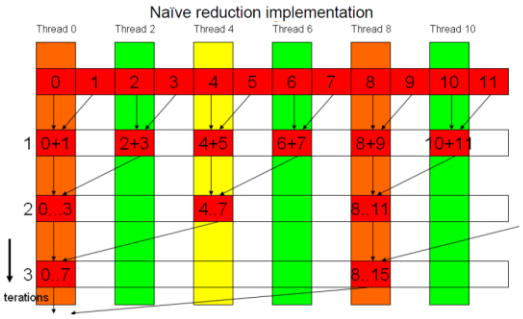
En general:

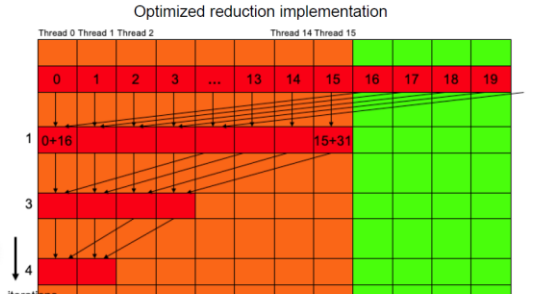
$$S(n) = \frac{T(1)}{T(n)} = \frac{n-1}{\log_2(n)} \qquad E(n) = \frac{S(n)}{n} = \frac{n-1}{n} \cdot \frac{1}{\log_2(n)}$$



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.

- b) 8.
- c) Todos, los hilos impares no hacen nada.
- d) Solo cuando llegas a tamaño inferior a 32.
- e) Cargamos primero de memoria al array compartido los datos y trabajamos en el array compartido.






```
__global__ void reduce0(float *d_in,float *d_out){
   __shared__ float sdata[THREAD_PER_BLOCK];

   //each thread loads one element from global memory
   unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
   unsigned int tid=threadIdx.x;
   sdata[tid]=d_in[i];
   __syncthreads();

   // do reduction in shared mem
   for(unsigned int s=1; s<blockDim.x; s*=2){
      if(tid%(2*s) == 0){
         sdata[tid]+=sdata[tid+s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```