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

SOLUCION:

a) Con 8X8, 64 threads per Block. Con 8 bloques (maximo por SM) se consiguen 512 threads simultaneous por SM, repartidos en 16 Warp. No se consigue llegar al máximo de warps de 24 (que corresponden a 24x32 = 768 threads)

Since each SM can take up to 768 threads, it can take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

- Para 16X16, 256 threads per Block. Con 3 bloques se consigue up to 768 threads alcanzando el máximo número máximo de warp que pueden correr simultáneamente en cada multiprocesador

,d it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

b) For 32X32, son 1024 threads per Block y no es posible porque se supera el máximo número de threads por bloque de 512.

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?

SOLUCION.

Si todos los bloques son del mismo Kernel, como es el caso de esta arquitectura.

El número de bloques de la aplicación es mayor de uno y se pueden ajustar a uno de los siguientes tamaños de bloque para conseguir los 768 Threads (24 warps):

384 Threads /Bloque 2 bloques
256 Threads /Bloque 3
192 Threads /Bloque 4
128 Threads /Bloque 6
96 Threads /Bloque 8

No son posibles 5 bloques ni 7.

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.

Solucion.

Cada 4 instrucciones un acceso a memoria:

W0-I1: 4c + W0-I2: 4c + W0-I3: 4c + W0-I4: 4c + W0-I4: 200 c……. + W0-I5: 4c + W0-I6: 4c + …

Para la latencias de 200 ciclos se necesitaran 200/4 = 50 Instrucciones de otros warp y como estos también tendrán accesos a memoria cada 4 instrucciones va a ser necesario cambiar de warp cada 4 instrucciones por el acceso a memoria , entonces 50/4 =12,5 -> 13 son los warp que se necesitan para conseguir ocultar la latencia de 200 ciclos

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?

SOLUCION

Cada Bloque de Thread necesita 4x1024 = 4096 bytes de memoria compartida

Limidado por la memoria compartida

1024 x 4 = 4096 bytes

cada bloque necesita 4K y como hay 64 K de Shared memoria , entonces se pueden lanzar solo 16 bloques.

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?

Inicialmente con 8K registros por bloque se podían lanzar 256K/8K = 32 (el número máximo de bloques que se puede lanzar.

Al incrementa MAX y gastar 32K registros por bloque se podían lanzar 256K/32K = 8 el número máximo de bloques que se puede lanzar. Siendo el nuevo límite en lugar de la memoria compartida.

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.

SOLUCION

Memoria compartida utilizada por bloque : 2 x 4 x block_size + 16 x 4

1024: 8x1024 + 64 = 8K + 64 , entonces en 16 K solo se puede lanzar 1 bloque
512: 8x512 + 64 = 4K + 64, entonces en 16 K solo se puede lanzar 3 bloques
256: 8x256 + 64 = 2K + 64, entonces en 16 K solo se puede lanzar 7 bloques
128: 8x128 + 64 = 1K + 64, entonces en 16 K solo se puede lanzar 15 bloques
64: 8x64 + 64 = 576 bytes entonces 16384/ 576 = 28,44 se pueden lanzar 29 bloques.

Block_size= num Threads/block	Blocks /SM	Treads/SM	
64	28	1792	
128	15	1920	
256	7	1792	
512	3	1536	
1024	1	1024	

La mejor opción es Block_size= num Threads/block = 128
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?

Block_size= num Threads/block	Blocks /SM	Treads/SM	
64	8	512	
128	8	1024	
256	7	1792	
512	3	1536	
1024	1	1024	

La mejor opción es Block_size= num Threads/block = 256

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?

Block_size = num Threads/block	Blocks /SM	Treads/SM	Register/block	Register/SM
64	8	512	64x9=576	4608
128	8	1024	128x9=1152	9216
256	7	1792	2304	16128
512	3	1536	4608	13824
1024	1	1024	9216	9216

In all cases, the total register usage is lower than 16384.

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?

Block_size = num Threads/block	Blocks /SM	Treads/SM	Register/block	Register/SM
64	8	512	64x10= 640	5120
128	8	1024	128x10 = 1280	10240
256	**6**	**1536**	2560	15360
512	3	**1536**	5120	15360
1024	1	1024	10240	10240

The highest occupancy will be 1536/ 2048 = 0,75 . It can be obtained with blocks of 256 or 512 threads.

Explanation:
The number of registers per thread is 10 in the second kernel. The configuration with 256 threads per block for the second kernel version will be able to allocate one less block per

SM (6) than for the first kernel version (7):

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.

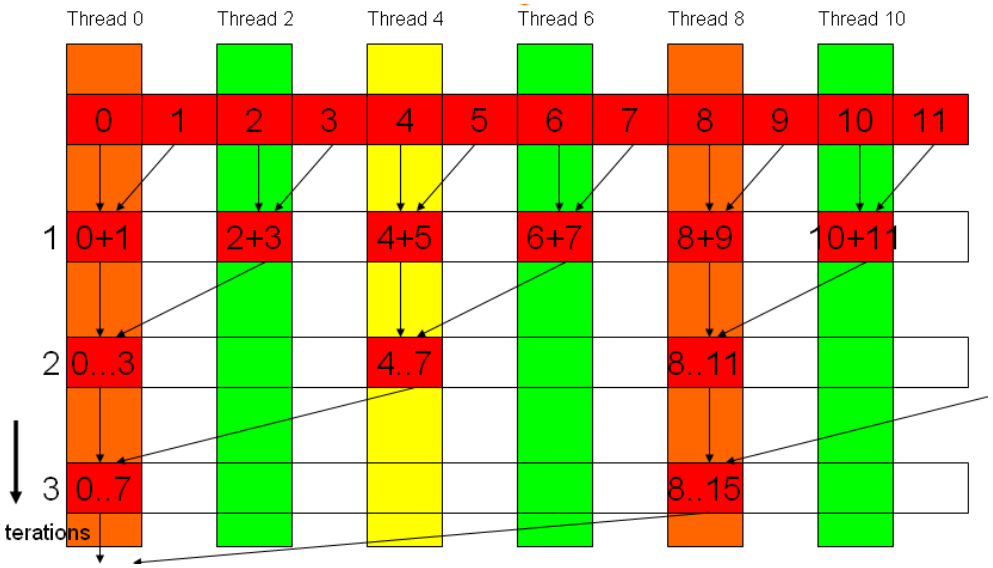
```
__global__
void matrix_mul_kernel(float* Md, float* Nd, float* Pd, const int cWidth)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  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_;
}
```

SOLUCION:

1. __syncthreads para garantizar que ha terminado la copia a la memoria compartida desde la memoria global

2.- __syncthreads para garantizar que ha terminado la acumulación de valores antes de pasarlos a la varible en memoria global.

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.

SOLUCION

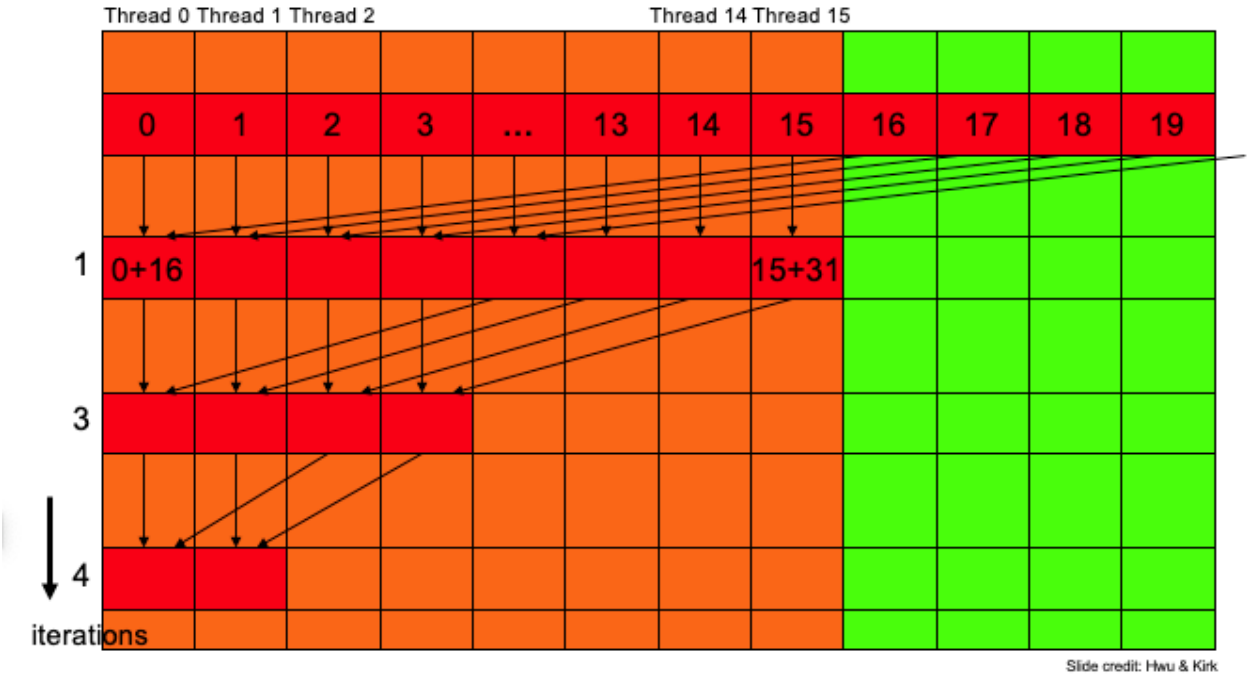
a) Los hilos impares no hacen nada de trabajo y cada iteración se reduce a la mitad el numero de hilos activos. Hasta la úlitma iteracción con stride = mayor potencia de dos con valor menor de blockDim.x.
En esa iteración la actividad es 1/bloqDimx

b) los warps programados tienen como minimo el 50% de hilos inactivos.
la eficiciencia E= S/n = 1/log n = con S= n/logn

c) Los hilos impares no hacen nada de trabajo y cada iteración se reduce a la mitad el numero de hilos activos. Hasta la úlitma iteracción con stride = mayor potencia de dos con valor menor de blockDim.x.
En esa iteración la actividad es 1/bloqDimx

d)

```
__shared__ float partialSum[]
unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1;  stride >> 1){

  __syncthreads();

        if (t < stride)
          partialSum[t] += partialSum[t + stride];

}
```



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);
    }
```

SOLUCION

the first warp corresponds to threads with these (x, y) indexes:

W0 (i,j)

```
(0,0), (1,0), ... (15,0),
(0,1), (1,1), ... (15,1).
```

All threads of a warp operate in a fully synchronized fashion. If one thread is reading `d[n*i + k]`, all threads of the warp are reading it simultaneously.

the **first warp of the first block**;

`float x = d[n*i + k];`

To set the value of `x`, the 32 threads in the warp will try to read the following array elements simultaneously:

```
d[0], d[1000], ..., d[15000],
d[0], d[1000], ..., d[15000].
```

This is **bad** from the perspective of the memory controller. The memory reads are likely to span 16 different cache lines, and we are only using 1 element per cache line

Incidentally, the memory access pattern is much better for the line
`float y = d[n*k + j];`

```
d[0], d[0], ..., d[0],
d[1], d[1], ..., d[1].
```

We are just accessing 2 different elements, both likely to be on the same cache line, so the memory system will need to do much less work here. The hardware is clever enough to realize that we are reading the same element many times.

here is a very simple solution: just swap the role of the indexes `i` and `j`:

```
__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*j + k];
        float y = d[n*k + i];
        float z = x + y;
        v = min(v, z);
    }
    r[n*j + i] = v;
}
```

Now on the line `float x = d[...]` the memory addresses that the warp accesses are good; we are accessing only two distinct elements:

```
d[0],    d[0],    ..., d[0],
d[1000], d[1000], ..., d[1000].
```

And on the line `float y = d[...]` the memory addresses that the warp accesses are also good; we are accessing one continuous part of the memory:

```
d[0], d[1], ..., d[15],
d[0], d[1], ..., d[15].
```

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?

SOLUCION:

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] -1 0 1 0 1 0 1 0 1
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8] 0 0 1 1 2 2 3 3 -1

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

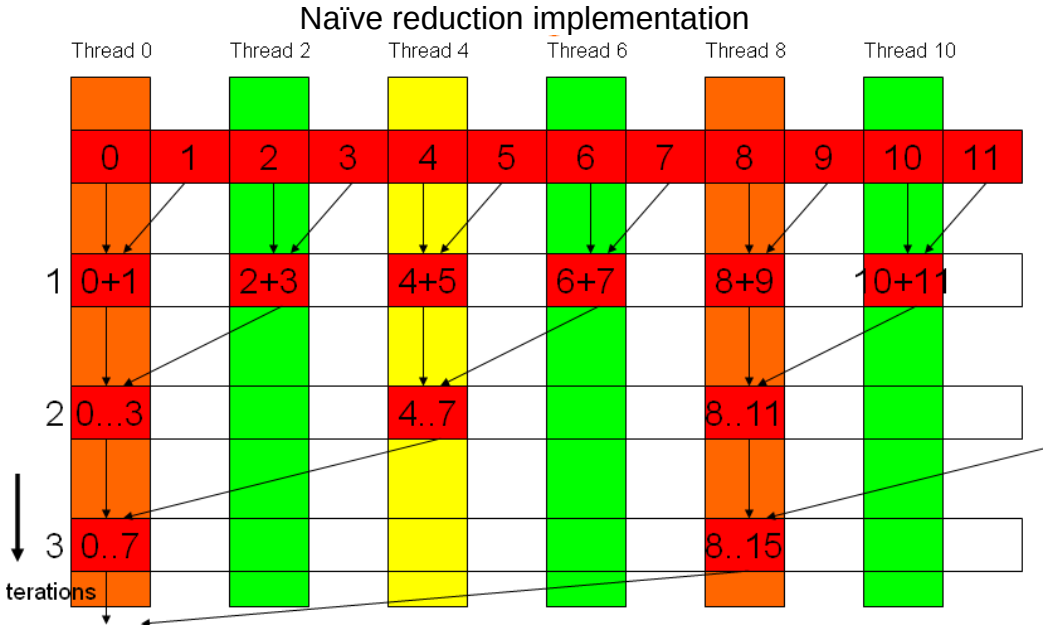
Kernel<<< dim3 (8,4,2), dim3(16,16) >>> (…)

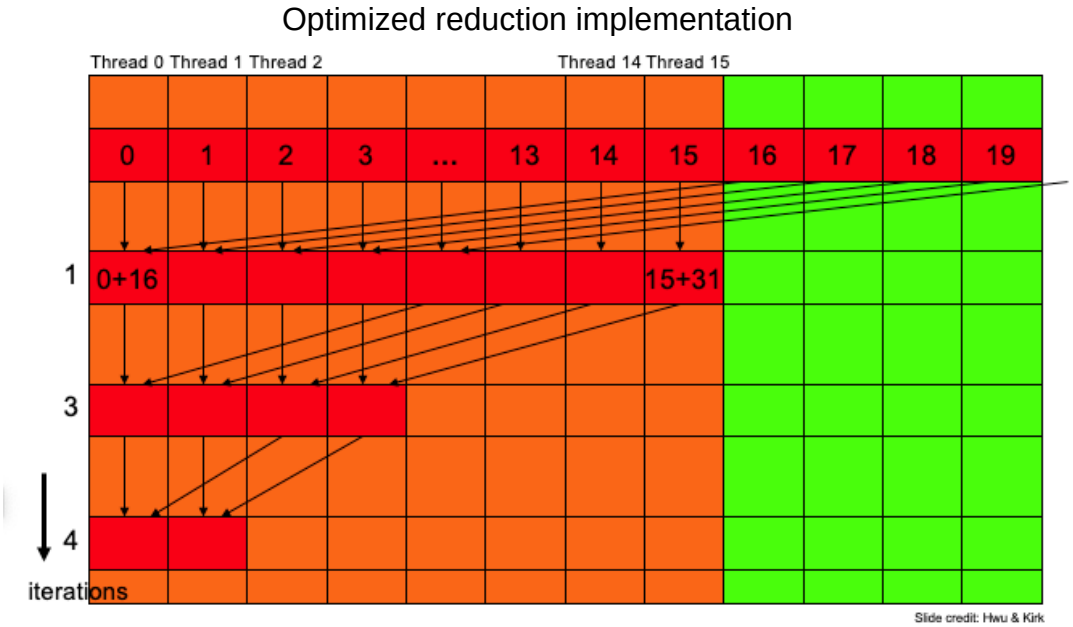
- ¿Cuántos bloques se están lanzando?
- ¿Cuántos threads por bloque?
- ¿Cuántos threads en total en el Stream Multiprocessor(SM)?

SOL

- ¿Cuántos bloques se están lanzando? 64
- ¿Cuántos threads por bloque? 256
- ¿Cuántos threads en total en el Stream Multiprocessor(SM)? 16364

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?

T serie (n-1)xTs
T paralelo = log (n) x Ts
S= n/log(n)
E=1/log(n)

b. Para el kernel denominado “naïve reduction”, ¿Cuantos pasos (iteraciones en la reducción) se necesitan?

8 steps – 128, 64, 32, 16, 8, 4, 2, 1

c. Explique que es la divergencia de threads e indique para la implementación “naïve reduction” cuántos pasos tienen divergencia indicando la razón.

8 steps – All, because not all threads in the warps are active

d. ¿Para la implementación denominada “optimized reduction” indique que pasos tiene divergencia de threads y cuáles no?

5 steps – When the number of active threads is less than warp size

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.

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

    //each thread loads one element from global memory to shared mem
    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];
}
```

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?

ceil(50000/2048) = 25

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?

1024 threads por bloque / 32 threads por warp = 32

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?

25 bloques x 1024 threads = 25600

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.

Block	Thread	First thread	Last thread
0	0	0	1024
0	1023	1023	2047
1	0	2048	3072
1	1023	3071	4095
23	0	47104	48128

23	1023	48127	49151
24	0	49152	50176
24	1023	50175	51199

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.

Si, inicialmente los dos if (i<n). Si nos basamos en los elementos que procesa cada bloque de la pregunta anterior, solo el bloque 24 procesa items >= 50000.
Linea 5 --> 50000 = Tid + 1024*24*2 --> Tid 848 (Warp 26 Block 24)
Linea 7 --> 50000 = Tid + 1024*24*2 + 1024 -> Tid = todos (todos los waprs) --> Todos los valores de I > 50000 por lo que no hay divergencia en esta linea.

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.

Al contrario de lo que se podría esperar, no hay más divergencia, ya que el comportamiento es el mismo. Mirar sección “Compare no-divergent warps” del profiling.
Problemas de alineamiento de memoria en el kernel que computa 2 elementos.
Menos threads por bloque
Una ventaja sería que se hace más trabajo por bloque

Respuesta detallada del ejercicio usando un profiler

Kernel con 1 solo elemento:

```
__global__ void vecAddKernel_1 (float* A_d, float* B_d, float* C_d, int n)
{
            int i = threadIdx.x + blockDim.x * blockIdx.x;
            if (i < n) { C_d[i] = A_d[i] + B_d[i]; }
}

vecAddKernel_1<<<ceil (n / 1024.0), 1024>>> (A_d, B_d, C_d, n);
```

Información de ejecución:

vecAddKernel

Section: Launch Statistics

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		25
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	25,600
Waves Per SM		0.62

WRN The grid for this launch is configured to execute only 25 blocks, which is less than the GPU's 40 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.

vecAddKernel_1

Section: Launch Statistics

Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		49
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	50,176
Waves Per SM		1.23

WRN If you execute __syncthreads() to synchronize the threads of a block, it is recommended to have more than the achieved 1 blocks per multiprocessor. This way, blocks that aren't waiting for __syncthreads() can keep the hardware busy.

Ocuppancy

vecAddKernel

Section: Occupancy

Block Limit SM	block	16
Block Limit Registers	block	4
Block Limit Shared Mem	block	16
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	96.40
Achieved Active Warps Per SM	warp	30.85

vecAddKernel_1

```
  Section: Occupancy
  ------------------------------------------------------------------------ --------------- ------------------------------
  Block Limit SM                                                                     block                           16
  Block Limit Registers                                                              block                            4
  Block Limit Shared Mem                                                             block                           16
  Block Limit Warps                                                                  block                            1
  Theoretical Active Warps per SM                                                     warp                           32
  Theoretical Occupancy                                                                  %                          100
  Achieved Occupancy                                                                     %                        95.02
  Achieved Active Warps Per SM                                                        warp                        30.41
  ------------------------------------------------------------------------ --------------- ------------------------------
```

Compare no-divergent warps

```
vecAddKernel(float*, float*, float*, int), 2022-Dec-16 08:50:21, Context 1, Stream 7
  Section: Command line profiler metrics
  ------------------------------------------------------------------------ --------------- ------------------------------
  sm__sass_average_branch_targets_threads_uniform.pct                                    %                        99.98
  ------------------------------------------------------------------------ --------------- ------------------------------

vecAddKernel_1(float*, float*, float*, int), 2022-Dec-16 08:50:21, Context 1, Stream 7
  Section: Command line profiler metrics
  ------------------------------------------------------------------------ --------------- ------------------------------
  sm__sass_average_branch_targets_threads_uniform.pct                                    %                        99.98
  ------------------------------------------------------------------------ --------------- ------------------------------
```

vecAddKernel

```
 Section: Source Counters
 ------------------------------------------------------------------------ --------------- ------------------------------
 Branch Instructions Ratio                                                              %                         0.05
 Branch Instructions                                                                 inst                        6,327
 Branch Efficiency                                                                      %                        99.98
 Avg. Divergent Branches                                                                                          0.01
 ------------------------------------------------------------------------ --------------- ------------------------------
```

vecAddKernel_1

```
 Section: Source Counters
 ------------------------------------------------------------------------ --------------- ------------------------------
 Branch Instructions Ratio                                                              %                         0.04
 Branch Instructions                                                                 inst                        6,263
 Branch Efficiency                                                                      %                        99.98
 Avg. Divergent Branches                                                                                          0.01
 ------------------------------------------------------------------------ --------------- ------------------------------
```

Memory

```
vecAddKernel(float*, float*, float*, int), 2022-Dec-16 08:52:21, Context 1, Stream 7
  Section: Memory Workload Analysis
  ------------------------------------------------------------------------ --------------- ------------------------------
  Memory Throughput                                                        Gbyte/second                        48.57
  Mem Busy                                                                               %                         7.13
  Max Bandwidth                                                                          %                        15.43
  L1/TEX Hit Rate                                                                        %                            0
  L2 Hit Rate                                                                            %                        36.58
  Mem Pipes Busy                                                                         %                         5.08
  ------------------------------------------------------------------------ --------------- ------------------------------
```