Computación de Altas Prestaciones
Seminario sobre GPGPUs
Sesión 4

Contenidos

Sesión 1 Teoría: Introducción a Computación en GPUs con CUDA.

Sesión 2:Compilación, conceptos básicos

Sesión 3: Programación de algoritmos “trivialmente paralelos”

Sesión 4: Uso de la memoria “Shared”. “Reducciones” en GPUs

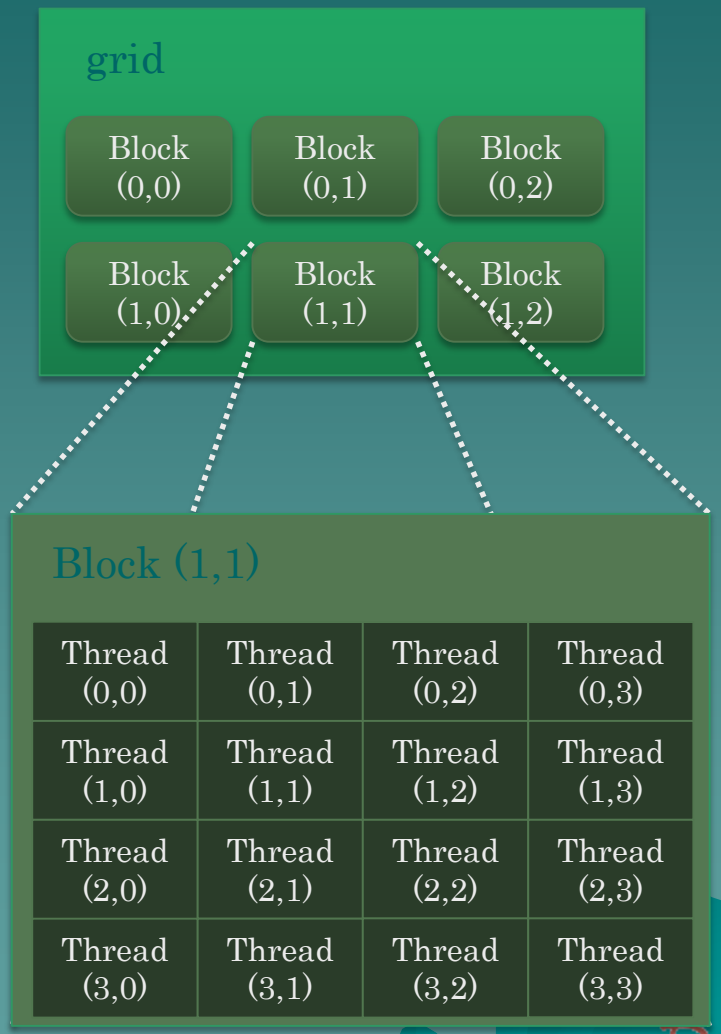
Sesión 5: Optimización, temas avanzados, librerías

Programación Paralela en CUDA: threads, bloques, grids

Los bloques pueden ser unidimensionales (como en los ejemplos anteriores), bidimensionales (como una "matriz" de threads) o incluso tridimensionales (como un "cubo de threads").

Los "grids" también pueden ser unidimensionales, bidimensionales o tridimensionales.

Por ejemplo, si el bloque es bidimensional, el número de fila y columna de thread se obtienen como threadIdx.x, threadIdx.y



Programación Paralela en CUDA: Sincronización, cooperación.

No todos los problemas son “trivialmente paralelos”

Cuando se lanza un kernel, hay que tener en cuenta varios aspectos:

1) Cada bloque de threads se ejecuta de forma independiente de los otros bloques, y no es posible garantizar que uno se ejecute antes que otro.

2) En general, los threads de un bloque también se ejecutarán en cualquier orden (aunque lo recomendable es lograr que se ejecuten totalmente en paralelo). Si es necesario, dentro del kernel se pueden sincronizar los threads con la función __syncthreads().

3) Todos los threads dentro de un bloque comparten una especie de memoria cache llamada memoria shared, de acceso más rápido que la memoria global. (memoria “normal”). Esta memoria es relativamente pequeña.

Programación Paralela en CUDA: Producto escalar

Vamos a usar como caso de estudio el producto escalar de dos vectores de números reales.

Si x=(x_1,x_2,x_3,x_4), y=(y_1,y_2,y_3,y_4), entonces el producto escalar

$x \cdot y^T$ es $x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$.

El resultado es un número real

Vamos a adoptar una estrategia parecida a sumar vectores o matrices: cada thread hace los productos que le tocan y los va acumulando en una variable ; el problema es que luego hay que sumarlos todos.

Programación Paralela en CUDA: Producto escalar

Estrategia de solución:
El cálculo del producto escalar se reparte, inicialmente, entre los bloques de threads que se lanzan. Si los vectores son de longitud N, y hay B bloques, cada bloque calculará el producto escalar de un “trozo” de vector con N/B componentes.

-Al acabar el cálculo de cada bloque, cada bloque obtiene un número como resultado parcial. El resultado final se obtiene sumando los resultados parciales de los B bloques.

-Como el número de bloques (y de resultados parciales) es relativamente pequeño no es eficiente usar la GPU para esta última fase: Los resultados parciales se envían a la CPU y allí se suman.

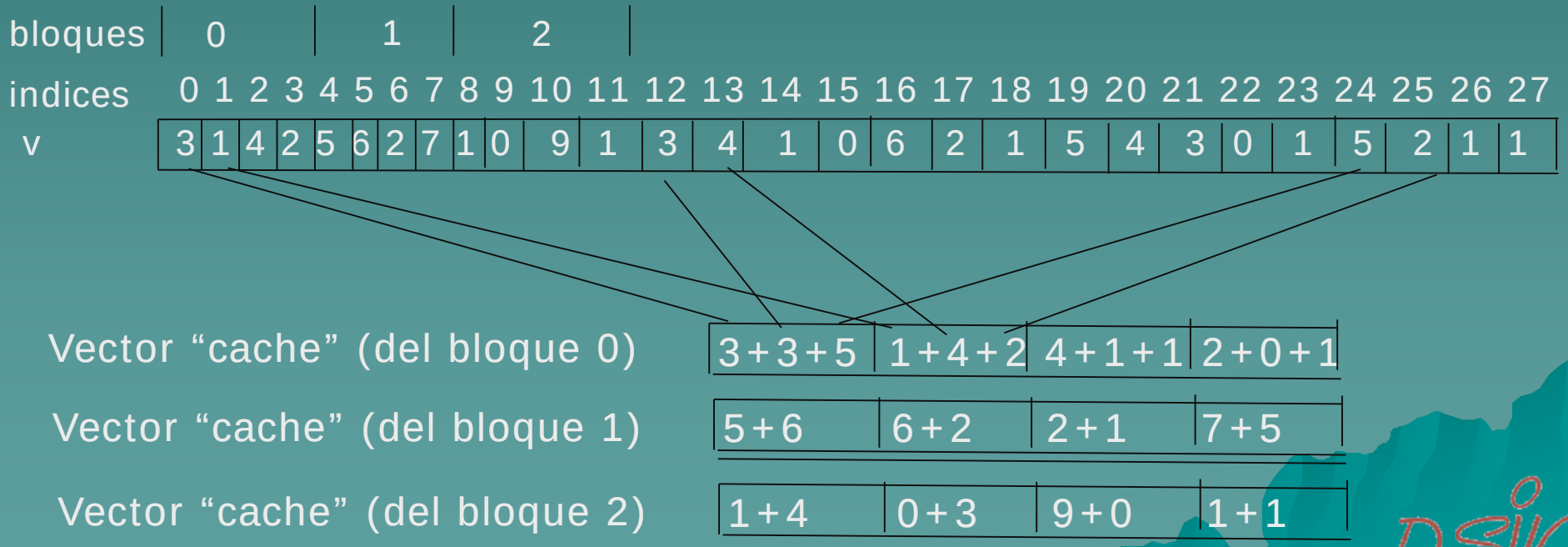
Programación Paralela en CUDA: Producto escalar

Implementación “simple”:

```
const int N=1024*20
const int threadsperBlock = 256
__global__ void producto_esc(float *a, float * b, float * sal)
{
 __shared__ float cache[ThreadsPerBlock];
 int tid=threadIdx.x+blockIdx.x * blockDim.x;
 int cacheindex=threadIdx.x;
 float suma=0.0,temp=0.0;
while (tid<N)
  {temp +=a[tid]*b[tid];
    tid+=gridDim.x * blockDim.x;
  }
  cache[cacheindex]=temp;
…//falta una parte, calcular la suma;
If (threadIdx.x==0)
    sal[blockIdx.x]=suma;
}
```

Programación Paralela en CUDA: Reducción

Supongamos que tenemos el vector v, queremos sumarlo, usando 3 bloques de 4 threads cada uno. Cada bloque tendrá un vector “cache” en memoria shared con 4 componentes, una por cada thread. El thread 0 del bloque 0 guardará en la posición 0 la suma de las componentes que le toquen:



Programación Paralela en CUDA: Producto escalar

En este código (incompleto), cada thread hace los cálculos para las posiciones de vector :

```
threadIdx.x+blockIdx.x * blockDim.x;
threadIdx.x+blockIdx.x * blockDim.x +gridDim.x * blockDim.x;
threadIdx.x+blockIdx.x * blockDim.x+2*gridDim.x * blockDim.x;
threadIdx.x+blockIdx.x * blockDim.x+3*gridDim.x * blockDim.x;
```

...

Para obtener el resultado final necesitamos:

1) Cada bloque de threads debe sumar todos sus resultados acumulados en el vector “cache” ➔ Cada bloque debe realizar una “reducción”

2) Una vez realizado esto, cada bloque habrá calculado “parte” del producto escalar; para completar el cálculo, hay que realizar otra suma (reducción) con todos los resultados parciales de cada bloque (en la CPU).

Programación Paralela en CUDA: Producto escalar

El problema que tenemos que resolver (sumar muchos elementos en paralelo) es una reducción: dado un vector con muchos elementos, realizar un cálculo que involucra a todos los elementos del vector pero cuyo resultados es un solo número (o unos pocos).

Hay muchos problemas de reducción, todos se tratan de la misma forma: suma, máximo, mínimo, producto, media, ...

Forma sencilla (e ineficiente) de hacer la reducción en un bloque de threads de una GPU: Hacerlo con un solo thread:

```
if (threadIdx.x==0)
   for (i=0;i<blockDim.x; i++)
     suma+=cache[i];
```

Sin embargo, esto también puede fallar; no hay garantía de que todos los threads hayan acabado y escrito su resultado parcial en el vector “cache”.

Programación Paralela en CUDA: Sincronización entre threads de un bloque

Cuando necesitamos garantizar que todos los threads de un bloque han alcanzado un cierto punto, podemos hacerlo con la instrucción

__syncthreads();

En nuestro ejemplo:

```
...
while (tid<N)
  {temp +=a[tid]*b[tid];
    tid+=gridDim.x * blockDim.x;
  }
  cache[cacheindex]=temp;

__syncthreads();

if (threadIdx.x==0)
   for (i=0;i<blockDim.x; i++)
     suma+=cache[i];
...
```

Programación Paralela en CUDA: Reducción en paralelo

Necesitamos calcular la suma

```
if (threadIdx.x==0)
   for (i=0;i<blockDim.x; i++)
     suma+=cache[i];
```

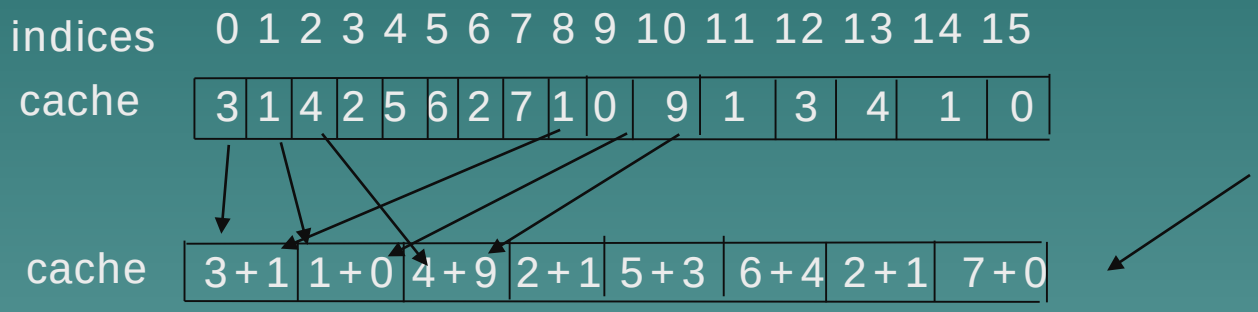
Pero en paralelo,usando cuantos mas threads mejor. Vamos a suponer que el número de threads del bloque es potencia de dos.

Entonces podemos hacerlo así:

```
int i=blockDimx.x/2;
while (i!=0)
    { if (cacheindex<i)
        cache[cacheindex]+=cache[cacheindex+i];
      __syncthreads();
      i=i/2;
     }
```

Programación Paralela en CUDA: Reducción en paralelo

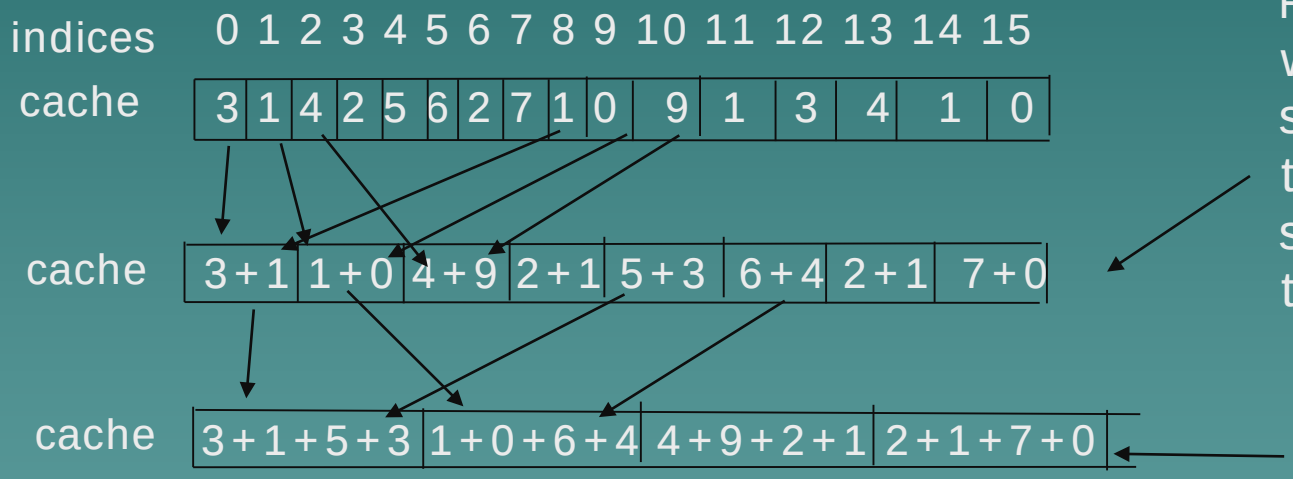
Veamos como funciona, supongamos que tenemos un bloque con 16 threads. El vector “cache” es de tamaño 16.



Primera etapa: El thread 0 suma lo suyo y lo del thread 8. El thread 1 suma lo suyo y lo del thread 9, etc:

Programación Paralela en CUDA: Reducción en paralelo

Veamos como funciona, supongamos que tenemos un bloque con 16 threads. El vector “cache” es de tamaño 16.



Primera iteración while: El thread 0 suma lo suyo y lo del thread 8. El thread 1 suma lo suyo y lo del thread 9, etc:

Segunda iteración while : El thread 0 suma lo suyo y lo del thread 4. El thread 1 suma lo suyo y lo del thread 5, etc:

Programación Paralela en CUDA: Reducción en paralelo

Cuando acaba este bucle la suma total del trozo de vector correspondiente a este bloque está almacenada en cache[0];

Entonces el thread 0 guarda el resultado en el vector de salida:

```
if (cacheindex==0)
    sal[BlockIdx.x]=cache[0];
```

Programación Paralela en CUDA: Ejercicios

1) Acaba de implementar el ejercicio de producto escalar de dos vectores (Puedes partir del archivo en poliformat, prod_esc_incompleto.c).

Experimenta tomando tiempos incluyendo los CUDAmemcpy, o sin incluirlos.

2) Haz un programa para calcular en la GPU las medias de las columnas de una matriz usando reducción. Cada bloque calcula la media de una columna, haciendo colaborar todos los threads del bloque. Hazlo a partir del archivo en poliformat media_matriz_incompleto.c

3) Haz un programa que calcule el producto Matriz-Vector