UNIVERSIDAD DE ALMERÍA

Computación en procesadores gráficos

Programación con CUDA
Ejercicio 8

José Antonio Martínez García
Francisco M. Vázquez López
Manuel Ujaldón Martínez
Ester Martín Garzón

Contenidos

- Arquitectura de las GPUs
- **Modelo de programación SIMT**
- **Claves computacionales para la programación de GPUs**
- **Programación con CUDA: Ejercicio 8**
- Supercomputación gráfica y arquitecturas emergentes

Programación con CUDA

- **Ejercicio 8**
  - Producto matriz densa por vector: u = A v
  - Dimensiones A: N x M, u: N, v: M
  - Inicializar A y v aleatoriamente
    - Con <stdlib.h>:
    - srand48(time(NULL)) para establecer semilla
    - drand48() genera un float [0, 1)
  - Realizar código secuencial para comprobar que el kernel es correcto, comprobar con una matriz p.ej 5 x 5
  - Una vez comprobado, comprobar que la suma de los elementos del vector obtenido mediante la CPU es el mismo valor que se obtiene con la GPU
  - Parámetros de entrada del programa:
    - N: Número de filas
    - M: Número de columnas
    - BLOCKSIZE: Tamaño del bloque 1-D
    - Verbose: Salida por pantalla si es 1
    - Iterations: Iteraciones del kernel

Programación con CUDA

- **Ejercicio 8**: Código secuencial

```
for (i = 0; i < N; i++) {
  u[ i ]=0;
  for (j = 0; j < M; j++)
      u[ i ]+=A[ i*M+ j ] * v[ j ];
}
```

Programación con CUDA

- Medir tiempos para:
  - Transferencia CPU->GPU
  - Ejecución del kernel
  - Transferencia GPU->CPU, recordar si es necesario usar puntos de sincronización
- Calcular ocupación y bandwidth
- Obtener la tabla de tiempos usando los dos mejores valores de BLOCKSIZE y para A=1000x2000, 2000x1000, 2000x5000 y 5000x5000
- Preguntas:
  - ¿ Para qué estructuras (u, A, v) se realiza un acceso coalescente a memoria ?
  - Comentar topología de bloques/threads usada:
    - ¿ Cuantos threads calculan una fila ?
    - ¿ Se podría usar más de 1 thread por fila ?. Ventajas/Inconvenientes
  - Precisión en los resultados. ¿ Hay diferencias ?, ¿ Por qué ?

Programación con CUDA

- **Mejoras a realizar**:
  - Usar la caché de texturas para almacenar el vector v
  - Almacenar la matriz por columnas ¿ Qué se gana con esta optimización ?
  - Usar la memoria shared para realizar productos parciales de cada fila y usar más threads por fila
  - ¿ Se te ocurre alguna optimización sobre la actualización de cada valor de u ?
- Repetir el estudio de tiempos realizado para las optimizaciones planteadas y comparar con la versión inicial
- Obtener los ratios más importantes de cuda visual profiler

Programación con CUDA

- **Ejercicio 8: Cuda Occupancy Calculator**
  - nvcc mv.cu –o mv -Xptxas=-v
  - 8 Registros + 48 bytes shared memory

	9500 GT	GTX 285
16	33 %	25 %
32	33 %	25 %
64	67 %	50 %
128	100 %	100 %
256	100 %	100 %
512	67 %	100 %

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT
- Iterations = 100, BLOCKSIZE = 128

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001619	0,001681	0,007593	0,018158
Kernel	2,540294	1,967700	14,099960	35,091710
GPU-CPU	0,000095	0,000030	0,000035	0,000438
Bandwith	0,630000	0,813500	0,567400	0,570000

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT
- Iterations = 100, BLOCKSIZE = 256

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001607	0,001602	0,007318	0,017925
Kernel	2,565286	1,967527	14,118784	35,254478
GPU-CPU	0,000027	0,000030	0,000032	0,000752
Bandwith	0,623900	0,813600	0,566700	0,567400

Programación con CUDA

- **Ejercicio 8**
  - Preguntas:
    - ¿ Por qué se obtiene mejor rendimiento para 2000x1000 que para 1000x2000 ?
    - ¿ Por qué el bandwidth es extremadamente bajo ?

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT usando caché de texturas
- Iterations = 100, BLOCKSIZE = 128

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001724	0,001645	0,007338	0,018124
Kernel	1,598312	0,965838	9,177806	22,942274
GPU-CPU	0,000023	0,000027	0,000029	0,000407
Bandwith	1,001300	1,657400	0,871800	0,871800

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT usando caché de texturas
- Iterations = 100, BLOCKSIZE = 256

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001675	0,001604	0,007431	0,018315
Kernel	1,596967	0,968404	9,177196	22,943222
GPU-CPU	0,000021	0,000405	0,000027	0,000035
Bandwith	1,002100	1,653000	0,871800	0,871800

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT usando caché de texturas y almacenamiento por columnas
- Iterations = 100, BLOCKSIZE = 128

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001627	0,001607	0,007414	0,018180
Kernel	0,368768	0,083871	0,413185	4,589941
GPU-CPU	0,000020	0,000021	0,000470	0,000408
Bandwith	2,170500	9,548000	9,682800	2,179100

Programación con CUDA

- **Ejercicio 8**: Resultados para 9500 GT usando caché de texturas y almacenamiento por columnas
- Iterations = 100, BLOCKSIZE = 256

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001622	0,001661	0,007400	0,018224
Kernel	0,368343	0,083784	0,413164	4,590678
GPU-CPU	0,000020	0,000021	0,000022	0,000040
Bandwith	2,173000	9,557900	9,683300	2,178800

Programación con CUDA

- **Optimización**: caché de texturas, almacenamiento por columnas, y 4 threads por fila
- Reserva dinámica de memoria shared
- **Código host**:

```
int smemsize = BLOCKSIZE * sizeof(float);

kernel<<<gridsize, blocksize, smemsize>>>(…);
```

Programación con CUDA

- **Código device**:

```
template <typename T>
struct SharedMemory
{  __device__ T*getPointer(){
  extern __device__ void error(void);
  error();
  return NULL; }  };

template <>
struct SharedMemory<float>
{  __device__ float* getPointer(){
  extern  __shared__ float s_float[];
  return s_float; }  };
```

Programación con CUDA

- **Declaración kernel**:

```
SharedMemory<float> smem;

float *sdata = smem.getPointer();
```

Programación con CUDA

- **Ejercicio 8: Cuda Occupancy Calculator**
  - nvcc mv.cu –o mv -Xptxas=-v
  - 8 Registros + smem: 64 bytes estática + dinámica (blocksize*4)

	Shared memory	9500 GT	GTX 285
16	128 bytes	33 %	25 %
32	192 bytes	33 %	25 %
64	320 bytes	67 %	50 %
128	576 bytes	100 %	100 %
256	1088 bytes	100 %	100 %
512	2112 bytes	67 %	100 %

Programación con CUDA

- **Ejercicio 8**: 9500 GT: caché de texturas, almacenamiento por columnas y 4 threads por fila
- Iterations = 100, BLOCKSIZE = 128

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001632	0,001632	0,007441	0,018292
Kernel	0,091818	0,082910	0,407398	1,011214
GPU-CPU	0,000019	0,000020	0,000021	0,000027
Bandwith	8,71720	9,611500	9,820400	9,891100

Programación con CUDA

- **Ejercicio 8**: 9500 GT: caché de texturas, almacenamiento por columnas y 4 threads por fila
- Iterations = 100, BLOCKSIZE = 256

N x M/sec	1000x2000	2000x1000	2000x5000	5000x5000
CPU-GPU	0,001633	0,001609	0,007577	0,018003
Kernel	0,091539	0,082948	0,407411	1,011507
GPU-CPU	0,000020	0,000020	0,000022	0,000028
Bandwith	8,743800	9,654200	9,820100	9,888200

Programación con CUDA

- **Ejercicio 8**: Cuda Visual Profiler
- Iterations = 100, BLOCKSIZE = 256, 2000x5000

	1ª versión	caché	col+cach	4 T
gputime	141344	91768	4146	4086
cputime	141362	91790	4170	4109
occupancy	1	1	1	1
gridSizeX	8	8	8	32
gridSizeY	1	1	1	1
blockSizeX	256	256	256	256
blockSizeY	1	1	1	1
blockSizeZ	1	1	1	1
StaSmemperblock	48	48	48	64

Programación con CUDA

- **Ejercicio 8**: Cuda Visual Profiler
- Iterations = 100, BLOCKSIZE = 256, 2000x5000

	1ª versión	caché	col+cach	4 T
Registersperthread	8	6	7	10
Sm_cta_launched	2	2	2	8
branch	80130	80098	80098	80648
divergentbranch	2	2	2	72
instructions	640507	800404	640388	642759
warp_serialize	0	0	0	0
cta_launched	4	4	4	16

UNIVERSIDAD DE ALMERÍA