Implementación paralela del algoritmo Shell Sort bajo OpenCL

Francisco Huertas Ferrer

Universidad Complutense de Madrid

28 de junio de 2012

Programación paralela en dispositivos gráficos

- La programación paralela en dispositivos gráficos intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general. Para ello utiliza el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.
- Las aplicaciones se diseñan para que utilicen numerosos hilos que realizan tareas independientes.
- OpenCL es un framework que permite a los programadores implementar algoritmos para su ejecución en GPU.
- OpenCL consta de un lenguaje de programación, un compilador y un conjunto de herramientas.

Shell Sort

- Algoritmo de ordenamiento se basa en una modificación del algoritmo de inserción en base a las siguientes observaciones:
  1. El Ordenamiento de inserción es eficiente si el algoritmo está "casi ordenado".
  2. El ordenamiento es ineficiente, en general, porque mueve los valores una sola posición.
- El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga comparaciones entre posiciones más separadas y lo produce que que los elementos se muevan más rápido hacia su posición esperada
- El algoritmo realiza múltiples pasos sobre los datos. Cada paso se hacen con tamaños de espacio o incrementos más pequeños.

Paso 1, un hilo por subconjunto, Implementación A

- Shell Sort se divide en pasos. Cada paso ordena un subconjunto de elementos independientes.
- En cada paso se lanza un hilo por subconjunto.
- Ejemplo para 100 elementos, incrementos de Sedgewick (1, 5, 19, 41):
  1. 41 hilos, 2-3 elementos por hilo.
  2. 19 hilos, 5-6 elementos por hilo.
  3. 5 hilos, 20 elementos por hilo.
  4. 1 hilos, 100 elementos por hilo.

Paso 2, varios hilos por subconjunto, Implementación B

- El número de hilos de la implementación A se reduce en cada fase y el número de elementos que ordena cada hilo es mayor.
- Para mantener constante el número de hilos lanzados, esta implementación lanza más de un hilo por subconjunto que lo ordenan parcialmente.
- Los datos no se ordenan por completo porque no existe comunicación entre los hilos que ordenan el mismo subconjunto de elementos.
- Ejemplo para 100 elementos, incrementos de Sedgewick (1, 5, 19, 41), cada hilo ordena al menos 4 elementos:
  1. 41x1 (41) hilos, 2-3 elementos por hilo.
  2. 19x2 (38) hilos, 4 elementos por hilo (último hilo de cada subconjunto ordena 1-2 elementos).
  3. 5x5 (25) hilos, 4 elementos por hilo.
  4. 1x25 (25) hilos, 4 elementos por hilo.

Paso 3, Corrección de la imp. B, Implementación C

- En la implementación B no existen comunicación entre los hilos.
- Se añaden comparaciones entre los extremos de los datos que gestiona cada hilo. Si no se encuentran bien colocados entre ellos se intercambian y vuelven a ordenarse cada subconjunto.
- Como los elementos están "casi ordenados", no se espera un número de repeticiones muy alto.

Paso 4, Copia en local de los datos, Implementación D

- Se intenta mejorar el rendimiento del algoritmo realizando copia privada de los datos que maneja cada hilo.
- Cuando cada hilo ordena sus elementos, copian el primer y último de sus datos en memoria global.
- Se comparan los datos de memoria global, y si es necesario se intercambian los valores, copiando estos en memoria privada.

Hardware y variables del algoritmo utilizado

	Equipo A	Equipo B	Equipo C
Tipo	Portatil	Sobremesa	Sobremesa
CPU	Intel Core i5 560M	Intel Core i5 650	Intel Core 2 Duo
GPU	Quadro NVS 3100M	Nvidia GT 320	GeForce 8600 GT
Núcleos CUDA	16	72	36
Mem. Principal	4 GB	4 GB	2 GB
Mem. Gráfica	512 MB	512 MB	256 MB
S.O.	Ubuntu 12.04	Debian 6	Debian 6
OpenCL	1.1 (Cuda 4.2.1)	1.1 (Cuda 4.0)	1.1 (Cuda 4.0)

Cuadro: Equipos Utilizados

Se han utilizado los incrementos de Sedgewick los cuales tienen un coste de $O(N^{4/3})$

Tiempos de ejecución en el equipo A

Elementos x repeticiones	Lineal	Imp-0	Imp-A	Imp-B	Imp-C	Imp-D
100 x 1	0.00	0.01	0.00	0.00	0.00	0.00
1000 x 1	0.00	0.63	0.15	0.00	0.00	0.00
10 x 100	0.00	0.14	0.04	0.04	0.07	0.04
100 x 100	0.00	18.25	0.24	0.08	0.14	0.08
1000 x 100	0.02	—	14.56	0.10	0.34	0.17
10000 x 100	0.30	—	—	0.73	1.31	1.57
100000 x 100	3.76	—	—	10.78	15.08	21.32
400000 x 100	19.08	—	—	58.49	69.07	95.02
1000000 x 100	55.74	—	—	183.98	180.93	—

- La ejecución en CPU es mucho más rápida. Posiblemente debido a que la GPU ofrece unas prestaciones muy bajas.
- Los tiempos de la implementación lineal el GPU (Imp 0) y implementación A en dejan de ser tratables con un número de elementos muy pequeño. No obstante, la implementación A ya obtiene mejoras con respecto a la implementación 0.
- La implementación C, en las últimas etapas, mejora los tiempos de la la implementación B, pese a tener que hacer repeticiones
- La implementación D, con copia de datos en memoria privada, aparentemente es más ineficiente.
- Cuanto más elementos tiene el vector, menor es la relación de tiempos entre la implementación en GPU y la implementación en CPU.

Número de lecturas y escrituras, Implementación C

Número de mínimo de elementos por hilo = 5

1000 x 1000 (Elementos x repeticiones)						
Incremento	929	505	209	109	41	19
Hilos / grupo	1	1	1	2	7	11
Grupos	929	505	209	109	41	19
Datos	1.07	1.98	4.48	4.58	3.48	4.78
Lecturas	0.11	1.4	6.2	9.7	17	19
Escritura	0.11	1.4	6.2	8.8	15	16
Pasadas	1	1	1	1.6	2.5	2.8

1000 x 1000		——	10000 x 100		
Incremento	5	1	——	5	1	——
Hilos / grupo	40	200	——	400	512	——
Grupos	5	1	——	5	1	——
Datos / hilo	5	5	——	5	19,5	——
Lecturas	32	43	——	128	150	——
Escritura	27	36	——	123	172	——
Pasadas	4.0	5.5	——	4,1	22,6	——

Cuadro: Número medio de lecturas, escrituras y pasadas por hilo

- Cuando se alcanza el número máximo de hilos por incremento, el número de veces que es necesario reordenar se ve penalizado.
- El número de lecturas y escrituras a memoria global en las primeras fases es muy baja. Todo lo contrario en las últimas fases.

Análisis del tiempo por fases

Elementos 500.000	Repeticiones 100	Max. Hilos 512	Min. Elementos 5	
Incr.	Lineal	Imp B	Imp C	Imp D
260609	0.60	2.34	2.81	7.46
146305	0.70	3.13	3.69	7.80
64769	0.95	3.82	4.46	8.42
36289	0.96	3.84	4.42	8.48
16001	1.26	4.45	4.99	8.99
8929	1.35	4.22	4.83	8.94
3905	1.44	4.82	5.36	9.43
2161	1.48	4.45	5.12	9.30
929	1.56	4.95	5.52	9.74
505	1.42	4.72	5.38	13.15
209	1.64	5.47	5.96	7.60
109	1.52	4.84	5.40	4.61
41	2.88	5.16	10.46	3.06
19	1.94	4.36	8.11	2.27
5	1.87	14.18	9.48	1.45
1	1.89	16.34	15.07	4.57
Total	24.39	91.74	102.26	115.99

Elementos 1.000.000	Repeticiones 100	Max. Hilos 512	Min Elementos 1
Incr.	Lineal	Imp B	Imp C
587521	1.11	5.03	5.98
260609	1.56	6.58	7.96
146305	1.84	7.15	8.53
64769	2.16	8.26	9.63
36289	2.60	8.24	9.53
16001	2.74	9.25	10.44
8929	3.13	8.74	9.95
3905	3.08	9.61	10.78
2161	3.02	9.21	10.43
929	3.18	9.90	11.24
505	3.41	9.57	16.60
209	3.71	6.10	11.48
109	3.38	3.99	8.63
41	7.61	5.36	8.45
19	4.25	4.36	8.70
5	3.77	20.97	11.66
1	3.94	43.67	14.70
Total	56.29	177.23	176.93

- La primera fase es más lenta en GPU que en CPU.
- La implementación B se ve perjudicada en las últimas fases del algoritmo.
- La implementación D, en las fases en las que hay un mayor número de lecturas y escrituras, es más eficiente.

Tiempos de ejecución en distintas maquinas

Elementos	Repeticiones	Max. Hilos	Min. Elementos
400.000	20	512	5
	Lineal	Imp. B	Imp. C
Equipo A	4.05	11.59	13.24
Equipo B	3.92	5.61	3.78
Equipo C	4.53	7.80	7.64

- La relación entre el número de núcleo y la velocidad de ejecución están inversamente relacionadas
- El mejor tiempo obtenido se consigue con la GPU del equipo C, esta GPU posee más núcleos pero sigue sin ofrecer altas prestaciones.
- No ha sido posible probar la implementación D en los equipos B y C.

Conclusión

- ShellSort es un algoritmo que para ordenar los datos, divide en cada fase el conjunto de datos en varios subconjuntos que ordena independientemente. Esta característica favorece su implementación en dispositivos gráficos.
- Sin embargo, las últimas fases, al reducir el número de subconjuntos, hacen que el rendimiento del algoritmo en paralelo no sea tan bueno.
- Las diferencias entre las fases también afectan negativamente al rendimiento paralelo de las implementaciones paralelas. Mientras que en las primeras fases, la cantidad de datos que se ordena cada subconjunto es muy pequeña y se encuentran muy separadas, según van avanzando las fases los datos son más y se encuentran mas cercanos.

Implementación final

- La implementación final no consigue mejorar de forma significativa el tiempo de ejecución en CPU.
- Los datos obtenidos no son definitivos debido a la diferencia de prestaciones entre las CPU y los dispositivos gráficos de los equipos utilizados.

- Usar distintas implementaciones depende de las fases del algoritmo.
- Mejorar la gestión de la memoria privada, local y general.
- Buscar modificaciones del algoritmo que hagan la implementación paralela más sencilla.