Paralelización en GPU del algoritmo Shell Sort

Francisco Huertas Ferrer

***Abstract*—El desarrollo de software está sufriendo un cambio de concepto en favor de sistemas concurrentes. Este cambio de concepto se debe en gran medida a necesidad de mejorar las prestaciones actuales, ya sea para aumentar la capacidad de computo, mejorar la productividad de las aplicaciones o disponer de medidas de respaldo de sistemas. Estos sistemas concurrentes obligan a un cambio de concepto a la hora de desarrollar nuevo software y programar algoritmos.**

El sistema de concurrencia a nivel de núcleo de computo que ofrecen las arquitecturas de cálculo paralelo presentes en las unidades de procesamiento gráfico (Graphical Process Unit, GPU), posee una gran potencia de computo paralelo debido al inherente grado de paralelismo que poseen el procesamiento gráfico.

Con la aparición de lenguajes de programación como OpenCL y CUDA que permite delegar trabajo a la GPU aparece la posibilidades de utilizarlas para realizar cómputo paralelo de datos. Este tipo de programación plantea un nuevo reto para programar de nuevo algoritmos y optimizarlos al paralelismo ofrecido por la arquitectura de estos dispositivos. Este reto conlleva también un cambio de mentalidad la hora de realizar las implementaciones.

En este trabajo se ha buscado la paralelización del algoritmo de ordenación Shellsort y su implementación del mismo bajo OpenCL.

***Index Terms*—OpenCL, CUDA, Shellsort, paralelo**

I. Introducción

A. Potencia en unidades de procesamiento gráfico, procesamiento paralelo

Durante la última década, las Unidades de Procesamiento Gráfico (GPUs) han sufrido una gran evolución ofreciendo una capacidad de procesamiento muy superior a las Unidades de Procesamiento Central (CPUs). Esta superioridad esta basada en la peculiaridad de las operaciones de procesamiento de imágenes, que se consiste, en muchos casos, en realizar la misma operación para cada uno de los puntos de una imagen.

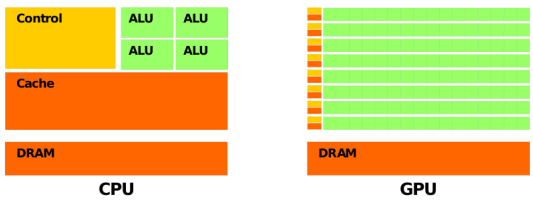

Figure 1: Diferencias arquitectónicas entre una CPU y una GPU

Aprovechando esta características, las GPUs basan su capacidad de computo en la utilización de gran cantidad de núcleos especializados que trabajan en paralelo. La arquitectura de estos núcleos es mucho más sencilla que las que tiene una CPUs como muestra la Fig.1. Esta arquitectura, por tanto, no es apta para programación general.

B. Unidades de procesamiento gráfico y programación general

La mejora de prestaciones de las GPUs ha aumentando el interés para el uso de estos dispositivos en otro tipo de programas. Este hecho ha producido la aparición de nuevos lenguajes de programación como CUDA y OpenCL que faciliten el uso de los dispositivos.

Actualmente están apareciendo implementaciones de algoritmos adaptados para su ejecución paralela que aprovechen la potencia de estos dispositivos gráficos. Estos algoritmos se caracterizan por ser altamente paralelizables pudiendo aprovechar al máximo la potencia de calcula de las GPUs.

II. Paralelización de Shellsort, Análisis

Los algoritmos con mayor nivel de paralelización ya se encuentra n implementado s, po r lo que encontrar un algoritmo que pueda implementarse de manera eficiente de forma paralela resulta complejo.

Al buscar un algoritmo para su paralelización se ha tenido en cuenta varias opciones, estudiando, entre los algoritmos de busqueda y ordenación, aquellos que no poseen todavía una implementaciones para OpenCL o CUDA. Entre estos algoritmos se ha elegido Shellsort debido al aparente caracter paralelo de sus operaciones de comparación entre elementos deslocalizados en su estructura de datos.

A. Shellsort

El algoritmo de ordenamiento Shellsort recibe su nombre en honor a su inventor Donald Shell. Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso. El algoritmo ha sido mejorado en diferentes implementaciones obteniendo rendimientos de $O(nlog^2(n))$ y $O(n^{3/4})$ en el peor caso.

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1) El ordenamiento por inserción es eficiente si la entrada está “casi ordenada”.
2) El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shellsort mejora el ordenamiento por inserción comparando elementos separados por un espacio

de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shellsort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

El rendimiento del algoritmo depende directamente de la separación entre los elementos comparados. Los mejores resultados hasta la fecha se obtienen usando los incrementos de Sedgewick con un coste de $O(nlog^2(n))$.

B. Requisitos de paralelización

La arquitectura particular de los dispositivos gráficos hacen que no todos los algoritmos sean elegibles para su paralelización. Estos requisitos indican cuales son los algoritmos que mejor se adaptan a la paralelización.

1) Operaciones: Para que un algoritmo pueda alcanzar un buen rendimiento paralelo es necesario que existan operaciones que se ejecuten muchas veces con diferentes datos y que la dependencia entre la misma operación con distintos datos no sea muy fuerte. Un ejemplo de esto es incrementar en uno el valor de un grupo de números. Esta operación no posee dependencias entre su ejecución con distintos datos y es repetida tantas veces como numero de datos existan.

2) Conjunto de datos: La paralelización de un algoritmo se basa en la repetición de una varias operaciones sobre un conjunto de datos. Para que sea interesante ejecutar el algoritmo en un dispositivo gráfico, debe ejecutarse sobre un grupo de datos lo suficientemente grande.

3) Representación de los datos: Los dispositivos gráficos están optimizados para tratar con imágenes y vídeos. Si los datos del algoritmo están distribuidos en estructuras que se asemejen a las utilizadas por imágenes y vídeos, el acceso a los mismos resultará mucho más sencillo. Estas estructuras son principalmente matrices de varias dimensiones.

Estos requisitos marcan los resultados positivos que se pueden obtener al paralelizar un algoritmo concreto

C. Análisis y valoración

El análisis de los requisitos en el algoritmo Shellsort se valora cada uno de los requisitos de paralelización:

1) Operaciones: El algoritmo se basa en el algoritmo de inserción, este no se caracteriza por ser un algoritmo paralelizable, sin embargo Shellsort se caracteriza por realizar comparaciones entre subgrupos de elementos. Los subconjunto de elementos son más numerosos y con menos elementos al principio del algoritmo. En la última etapa del algoritmo, existe un solo subconjunto con todos los elementos.

Los primeros pasos de Shellsort, al existir subconjuntos independientes de datos a comparar, la paralelización es sencilla, sin embargo en los últimos pasos. Al existir pocos subconjuntos es necesario buscar una forma de paralelizar de forma eficiente el algoritmo de inserción para listas de elementos "casi ordenadas".

2) Conjunto de datos: Al ser un algoritmo de ordenación, el conjunto de datos es lo suficientemente grande para realizar una implementación paralela del algoritmo.

3) Representación de los datos: La representación de los datos en una primera instancia puede favorecer una implementación paralela. Los datos se ordenan en una lista y los subconjuntos de datos se encuentran a distancias fijas.

Sin embargo hay características que van a afectar negativamente a este aspecto. El primero es que el tamaño de los subgrupos va a ser muy diferente en los diferentes pasos del algoritmo. Esto va a producir que la implementación tenga que adaptarse a este cambio.

Otro aspecto que puede afectar negativamente al algoritmo es que los elementos de cada cada subgrupo tiene sus elementos muy deslocalizados.

D. Conclusión del análisis

Shellsort, pese a poseer características que aparentemente lo hacen paralelizable, tiene aspectos mermarán la implementación paralela que se realice y, por tanto, la implementación no se beneficiara de la paralelización tanto como otros algoritmos. Esta puede ser la causa por la cual no exista aún una implementación de Shellsort para OpenCL o CUDA.

III. Paralelización de Shellsort, Implementación

La implementación paralela de Shellsort se ha realizado por fases. Cada una de las fases se ha centrado en un aspecto de la paralelización. También se ha hecho una implementación lineal para CPU y otra para GPU para comprar los resultados.

El código de la GPU se ha realizado en OpenCL. Se ha utilizado C para escribir el código que prepara el contexto OpenCL y lanza los hilos que se ejecutan en la GPU. La compilación se ha hecho en Linux. Los incrementos utilizados en el algoritmo son los incrementos de Sedgewick.

Por último, no todas las implementaciones ordenan por completo el vector de elementos.

A. Implementación lineal base, Shellsort 0

Esta implementación se ha utilizado como base para las siguientes implementaciones. Aunque se ejecute en la GPU el código se ejecuta de manera lineal.

El código que se ejecuta en la CPU es el encargado para prepara el contexto para la ejecución del código OpenCL. También se encarga de calcular los distintos incrementos.

El código para que se ejecuta en la GPU es el encargado de ordenar el subconjunto de elementos del vector que recibe. Los parámetros necesarios para realizar la llamada son:

- El propio vector de elementos. En dicho vector están todos los elementos del vector (*vector*)
- El tamaño del vector (*size*)
- El primer elemento que hay que ordenar (*offset*)

- La separación entre los elementos que hay que ordenar (*incremento*)

La ejecución del código OpenCL se realiza de manera secuencial para cada incremento y offset y se realiza en una dimensión y un único hilo por dimensión.

Esta implementación ordena los elementos del vector por completo

B. Paralelización de los incrementos, Shellsort A

Este nivel de paralelización es el más fácil de realizar. A la hora de lanzar los hilos en la GPU se lanzan simultáneamente un hilo por cada subconjunto de elementos. Por cada fase del algoritmo se lanzan n hilos donde n es el valor del incremento de Sedgewick para esa fase, en concreto las llamadas se realizan en una única dimensión de n elementos y no existen comunicación entre los hilos y por lo tanto cada hilo se ejecuta en un grupo distinto. Las fases se siguen ejecutando de manera secuencial.

En la implementación OpenCL se sustituye el parametro offset que indicaba el subconjunto que se ordenaba por el id global del hilo en la única dimensión existente.

Esta paralelización, para las primeras fases del algoritmo obtiene muy buenos resultados, sin embargo, en las ultimas fases el algoritmo no obtiene grandes resultados ya que el numero de hilos lanzados es muy pequeño.

Tomando los incrementos de Sedgewick (1, 5, 19, 41, 109, ...), para una lista de 100 elementos cada fase posee las siguientes características:

- 1º: 41 hilos, 2-3 elementos ordenados por hilo.
- 2º: 19 hilos, 5-6 elementos ordenados por hilo.
- 3º: 5 hilos, 20 elementos ordenados por hilo
- 4º: 1 hilos, 100 elementos ordenados por hilo

Esta implementación ordena los elementos del vector por completo.

C. Paralelización del algoritmo de inserción, Shellsort B

Con la base del apartado anterior se ha buscado una forma de mejorar el rendimiento cuando el número de subconjunto es pequeño. Para realizar esto se han lanzado un número variable de hilos por subconjunto que son los encargados de ordenar un pequeño grupo de elementos pertenecientes a dicho subconjunto.

La implementación lanza los hilos en dos dimensiones:

- La primera representa el subconjunto que ordena cada hilo.
- La segunda representa que parte del subconjunto ordena.

En esta implementación no es necesario clasificar los hilos por grupos, sin embargo, por claridad y se han agrupado los hilos en grupos que ordenan el mismo subconjunto de elementos del vector. De igual manera es aconsejable determinar un número máximo de hilos por subconjunto. este valor depende del número de elementos del subconjunto y nunca puede superar el tamaño máximo por dimensión por grupo de trabajo.

Tomando de nuevo los incrementos de Sedgewick (1, 5, 19, 41, 109, ...), para una lista de 100 elementos y estableciendo como máximo de hilos por grupo una cuarta parte de los elementos por grupo, las fases del algoritmo tienen las siguientes características:

- 1º: 41x1 hilos (41 en total), 2-3 elementos ordenados por hilo.
- 2º: 19x1 hilos (19 en total), 4-5 elementos ordenados por hilo.
- 3º: 5x5 hilos (25 en total),4 elementos ordenados por hilo.
- 4º: 1x25 hilos (25 hilos en total), 4 elementos ordenados por hilo.

Este algoritmo mejora notablemente el rendimiento con respecto a la versión anterior. Sin embargo esta implementación no ordena el por completo el vector de elementos, ya que no existe comunicación entre los hilos que ordenan el mismo subconjunto de elementos.

D. Variante de Shellsort, Shellsort C

Esta versión únicamente cambia el orden en que se ejecuta el algoritmo de inserción. En el apartado anterior se ejecuta del elemento menor al mayor. En este se ejecuta del mayor al menor. Al obtener resultados similares esta versión se ha descontinuado.

E. Corrección del algoritmo de inserción, Shellsort D

Esta corrección busca que el algoritmo planteado en Shellsort B ordene los elementos

La paralización Shellsort B deja los elementos ordenados por hilo. Para ordenar todos los elementos de un subconjunto se comparan el elemento mayor y menor de los hilos consecutivos. En caso de que no estén ordenados se intercambian los valores y comienza de nuevo a ordenarse cada hilo internamente.

Aparentemente esta forma de ordenar no es muy eficiente, sin embargo, se ha optado por esta versión porque los elementos de cada fase se encuentran más cerca de su posición real y, por lo tanto no, habrá hacer hacer muchas repeticiones.

Se ha planteado realizar una modificación de realizando más de un cambio de elementos entre hilos por pasada. Al final se descartó por no conseguir resultados óptimos.

Otra mejora que se planteó en este apartado es desactivar hilos que, aparentemente no necesiten volver a ordenar, sin embargo, al realizar un cambio en cualquiera de los elementos, cualquier hilo es sensible de necesitar reordenar sus elementos.

Por último añadir que se han incluido optimizaciones a la hora de traer los elementos de memoria. Estas mejoras intentan hacer el mínimo número de lecturas y escrituras en memoria global del dispositivo.

F. Uso de memoria local, Shellsort E y F

Un de los problemas que suele tener las implementaciones de algoritmos para OpenCL o CUDA es la gran

cantidad de tiempo que se pierde accediendo a memoria general. Como se podrá ver en la sección siguiente, el numero de accesos a memoria en la implementación Shellsort D puede llegar a ser muy alta. Para evitar esto se ha intentado mover el vector de memoria general a memoria privada.

El primer problema que se ha tenido que solventar a sido que en las últimas fases, que generalmente son las más lentas, tienen una cantidad tan grande de datos que es imposible almacenar en memoria toda el vector. Para solventar este problema se ha hecho que cada hilo trabaje en un grupo distinto. La pega de esto es que ahora todos los hilos tienen que esperarse para comprobar si ha habido cambios en su subconjunto de datos. Cuando un hilo ha terminado de ordenar sus datos realiza la copia en memoria general del mayor y menor de sus datos. Una vez hecho esto se comprueban los datos entre hilos para saber si hay que hacer intercambio.

Estas implementaciones, sin embargo, no ha mejorado nada el rendimiento con respecto a las versiones anteriores. Los principales problemas que se han encontrado es que al variar tanto las dimensiones tan variables con las que se trabaja.

IV. Estudio

Las pruebas que se han hecho para comprobar las distintas implementaciones se han basado en pruebas empíricas obtenidas al ejecutar el código en diferentes equipos.

Para comprobar

A. Hardware utilizado

Todas las implementaciones se han probado en al menos un equipo. Adicionalmente, las mejores implementaciones se han probado en dos equipos más. Las especificaciones de los equipos se encuentran en el apéndice A.

Una de las principales pegas de las pruebas realizadas es la imposibilidad de probarlo en dispositivos gráficos de alto rendimiento. Las pruebas se han hecho principalmente en el Equipo A. El dispositivo gráfico de este equipo solo dispone de 16 núcleos CUDA cuando los dispositivos actuales de gama alta disponen de miles de núcleos. Por otro lado el procesador es, con respecto a la tarjeta gráfica, más avanzado.

B. Varios tiempos un solo equipo

Las pruebas de rendimiento de las diferentes implementaciones se han realizado en la equipo A. La tabla I muestra los resultados de ejecutar el algoritmo con distintos números de trupo para cada una de las implementaciones. Para que los datos sean más fiables se han realizado más de una repetición por configuración.

Entre los datos obtenidos hay que destacar que el algoritmo más rápido es el ejecutado en la CPU. Una de las causas es la poca potencia del dispositivo gráfico con respecto de la CPU.

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

Table I: Tiempos de ordenación en el equipo A

Analizando los tiempos paralelos, lo primero que se puede observar son los resultados es como la implementación 0, la cual realiza la ejecución del algoritmo sin paralelizar en la GPU. Esta implementación deja de ser viable en seguida obteniendo resultados superiores al minutos con 1.000 datos y 100 repeticiones. Los resultados obtenidos en la implementación A, que paraleliza los subconjuntos de elementos en cada fase, mejoran los tiempos de la implementación 0, aunque con 10.000 elementos no los resultados dejan de ser óptimos.

La implementación B y C, que obtienen tiempos muy similares son las primeras que obtienen tiempos aceptables. Esta obtiene los mejores tiempos porque incluye una nueva paralelización que consiste en dividir la organización de cada subconjunto. Esta implementación sin embargo no ordena completamente el vector, ya que no existe comunicación entre los hilos que ordenan el mismo subconjunto de elementos. La implementación representa el mayor grado de paralelización ya que aparentemente no es posible dividir más el trabajo de cada hilo.

La implementación D realiza las mejoras necesarias para que la implementación B ordene realmente los elementos. Esta implementación es aparentemente más lenta que la B sin embargo la diferencia de tiempos entre ellas disminuye al aumentar el número de elementos, llegando a conseguir mejores tiempos al ordenar vectores de un millón de elementos. Esto puede deberse a que en cada fase, la implementación D deja más ordenado el vector.

Otra observación que hay que hacer es que la relación entre el tiempo que tarda en acabar la implementación lineal y la D es menor cuanto más elementos tiene el vector.

Las últimas implementaciones, E y F obtienen también tiempos similares entre ellos. Estas realizan copias a memoria de cache para intentar mejorar los tiempos de ejecución. Sin embargo los tiempos no mejoran con respecto a las implementaciones anteriores. Esto posiblemente es debido a toda la lógica que añade la implementación al re-

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 / hilo	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	—	512	512	—
Grupos	5	1	—	5	1	—
Datos / hilo	5	5	—	3,9	19,5	—
Lecturas	32	43	—	128	150	—
Escritura	27	36	—	123	172	—
Pasadas	4.0	5.5	—	4,1	22,6	—

Table II: Número medio de lecturas, escrituras y pasadas hilo

alizar la copia. Entre los detalles de la implementación hay que señalar que es necesario indicar en la implementación OpenCL la cantidad de memoria local que se va a necesitar y esto es uno de los problemas de estas implementaciones ya que cada thread dispone de una cantidad máxima de memoria que puede disponer para sus datos. De hecho la implementación no es capaz de ordenar más de un millón de elementos.

La tabla tabla I muestra los resultados de distintas ejecuciones de las diferentes implementaciones. Las implementaciones 0 y A muestran como al no estar paralelizado o tener una paralelización bastante pobre, los resultados enseguida se disparan. Por otro lado los tiempos nunca mejoran pero la relación entre el tiempo de ejecución lineal y el tiempo de ejecución paralela (Imp-D) es cada vez más pequeña. Hay que señalar que la implementación D consigue los mejores tiempos en la última ejecución. La Implementación E, que realiza copia en memoria local de los datos del vector, obtiene tiempos inferiores a la implementación D, esto se debe posiblemente a la complejidad de la lógica necesaria para controlar la copia local de los datos.

Los tiempos de la Implementación C no se han puesto por ser inferiores similares a la implementación B al igual que las implementaciones E y F. También indicar que la implementación E y F es necesario definir manualmente en la implementación OpenCL la cantidad de datos en local que va a haber.

C. Numero de lecturas y escrituras

En el apartado anterior se ha observado como realizar una copia local de los datos para realizar la ordenación no resulta efectiva. En este apartado se han analizado el número de lecturas, escrituras medias por hilo de la implementación D y los resultados se muestran en la tabla II.

El número de pasadas indica la cantidad de veces que un algoritmo tiene que reordenar sus elementos.

Para 1.000 elementos, el número de escrituras aumenta alcanzando más de 30 lecturas y escrituras en la última fase o lo que es lo mismo, aproximadamente 8 lecturas y 7 escrituras por dato en la última fase. Aunque no se un dato muy bueno se compensa con las pocas lecturas / escrituras de las primeras fases del algoritmo. Este dato se debe a que cada vez hay más hilos por subconjunto y los subconjuntos son más grandes. El número medio total de lecturas y escrituras por pasada y dato es aproximadamente 4 que, no es un valor muy negativo. El número óptimo de lecturas y escrituras es aproximadamente 2.

Para 10.000 elementos, se puede observar como en las dos ultimas fases, mientras la media de datos por hilo no sea superior a 4, el número de pasadas se mantiene en 4, sin embargo cuando se llega al máximo de hilos por grupo de trabajo, 512, el número de pasadas que hay que realizar también aumenta, con estos datos se puede deducir que el número de pasadas está directamente relacionado con el número de datos que ordena cada hilo. También se puede observar que la relación entre el número de lecturas / escrituras por dato y el número de hilos es, en la última pasada, muy inferior a la fase anterior. Este resultado te hace planea la duda de cual es la cantidad óptima datos que cada hilo debe ordenar y cuantos hilos como máximo deben lanzarse por subconjunto de elementos en cada fase. En análisis de la siguiente sección terminará se analizarán la diferencia de tiempos cambiando estos datos.

D. Análisis de los tiempos por fases

En la tabla III y la tabla IV se analizan los tiempos para cada una de las fases. Los parámetros que se comparan son el número de hilos máximos por subconjunto y el número mínimo de elementos que ordena cada hilo. Hay que señalar que si el número de elementos que tiene un subconjunto es inferior mínimo de datos por hilo, prevalece haber al menos un hilo por subconjunto de elementos.

Las pruebas se han realizado con dos configuraciones, la primera configuración se realiza con el mayor número de elementos posibles para la implementación D, un millón. Una segunda configuración se realiza con el mayor número posible para la implementación E, medio millón.

Las pruebas tienen dos parámetros, el número mínimo de datos por hilo y el número máximo de hilos por subconjunto. El primero afecta principalmente a las primeras fases, valores altos producen que estas fases tengan pocos hilos por subconjunto. Este valor deja de tener sentido cuando la relación entre el número de datos de un subconjunto y el número de hilos aumenta y es esta relación es la que indica cuantos datos va a ordenar cada hilo a partir de entonces.

Con los resultados obtenidos, se puede determinar que el número de datos mínimo de datos por hilo no afecta a la velocidad final mientras que no sea una cantidad muy grande.

Se puede observar también como la implementación B ofrece valores más dispares como puede verse en las ejecuciones donde solo cambia el parámetro de número de datos mínimo por hilo.

También puede observarse en los resultados obtenidos como los tiempos de ordenación de cada una de las fases

Elementos 1.000.000	Repeticiones 100	Max. Hilos 512	Min Elementos 1
Incr.	Lineal	Imp B	Imp D
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

Elementos 1.000.000	Repeticiones 100	Max. Hilos 256	Min Elementos 2
Incr.	Lineal	Imp B	Imp D
587521	1.17	5.07	5.81
260609	1.57	6.56	7.98
146305	1.83	7.06	8.40
64769	2.20	8.46	9.73
36289	2.55	8.17	9.41
16001	2.79	9.37	10.41
8929	3.14	8.50	9.93
3905	3.09	9.82	10.87
2161	3.03	9.06	10.41
929	3.20	10.07	11.16
505	3.39	9.55	10.73
209	3.72	8.97	16.52
109	3.46	5.84	11.32
41	7.82	6.85	9.42
19	4.33	4.93	8.48
5	3.83	21.52	10.95
1	3.89	16.74	13.62
Total	56.83	157.66	177.36

Table III: Tiempo de ejecución por fases

Elementos 500.000	Repeticiones 100	Max. Hilos 512	Min. Elementos 5	
Incr.	Lineal	Imp B	Imp D	Imp E
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

Table IV: Tiempo de ejecución por fases

	Implementación		
Equipo	Lineal	B	D
Equipo A	4.05	11.59	13.24
Equipo B	4.53	7.80	7.64
Equipo C	3.92	5.61	3.78

Table V: Tiempo de ejecución en distintos equipos

son constantes para la implementación lineal y D, la implementación B pese a ser más rápida que la D, empeora sensiblemente en las últimas fases. Una de las causas puede ser el hecho de que los subconjuntos, después de cada fase en la implementación B, no están realmente ordenados.

Las ejecuciones en las que el parámetro máximo número de hilos por subconjunto empeoran el rendimiento en las últimas fases, en concreto con hasta 64 hilos por subconjunto. Con GPUs que dispongan más núcleos, este dato será posiblemente más decisivo.

Por último hay, la configuración con medio millón de datos, donde se puede ejecutar la implementación E, se puede observar como, al hacer una copia en cache de los datos, el rendimiento del algoritmo en las fases finales del algoritmo es mejor debido a que se producen un mayor número de lecturas y escrituras por dato.

E. Tiempos en distintos equipos

Por últimos se han podido ejecutar tres implementaciones en tres equipos distintos, esto ha permitido comparar con dispositivos gráficos más potentes. Las pruebas se han limitado a las implementaciones Lineal, Shellsort B y Shellsort D. Las pruebas se han realizado con 400.000 elementos y 20 repeticiones. El número máximo de hilos por subconjunto se ha establecido a 512 y cada hilo ordena al menos 5 elementos. los resultados se encuentran en la tabla V.

La primera conclusión que puede obtenerse es que el algoritmo depende del número de núcleos que el dispositivo posea. El dispositivo gráfico del equipo B, sin ser más potente, posee el doble de núcleos y reduce a la mitad el tiempo de ejecución del algoritmo. El equipo C posee 5 veces más núcleos y reduce el tiempo de ejecución casi en una quinta parte.

El otro dato positivo que se puede observar es que en la ejecución de la implementación D en el equipo C mejora ligeramente el tiempo de la implementación lineal.

V. Conclusión

Los datos obtenidos en las ejecuciones de las diferentes implementaciones, sin ser concluyentes, debido a la potencia de los dispositivos gráficos, hacen pensar las implementaciones paralelas de este algoritmo pueden mejorar los tiempos de ordenación con respecto a la ejecución del mismo en una CPU. Sin embargo esta mejora no es tan significativa como los resultados obtenidos con otros algoritmos.

Entre las causas se encuentra la deslocalización de los datos que gestiona cada hilo y la diferencia de cantidad

de datos que manejan cada subconjunto en las diferentes fases, sobre todo la primera y la última.

También se puede observar como, el algoritmo ofrece mejores resultados cuantos más datos gestiona.

VI. Trabajo futuro

Entre las mejoras que se pueden realizar a la implementación paralela plateada se encuentra realizar una implementación dependiente de los incrementos y diferentes implementaciones para cada incremento. Al existir pocas fases, para un millón de elementos, 17 fases, la ganancia que se obtendrá con esto, sera mucho mayor. Además, esta implementación permitirá que al copia en local de los datos que cada hilo ordena, sea mucho más eficiente.

Appendix A
Equipos utilizados

A. EquipoA

CPU: Intel Core i5 560M
GPU: Nvidia Quadro NVS 3100M (16 Núcleos CUDA)
Memoria principal: 4 GB.
Memoria Gráfica: 512 MB.
Sistema operativo: Ubuntu 12.04
OpenCL: 1.1 (Cuda 4.2.1)

B. EquipoB

CPU: Intel Core i5 650
GPU: Nvidia GT 320 (72 Núcleos CUDA)
Memoria principal: 4GB.
Memoria Gráfica: 512 MB.
Sistema operativo: Debian 6
OpenCL: OpenCL 1.1 (Cuda 4.0)

C. EquipoC

CPU: Intel Core 2 Duo
GPU: Nvidia GeForce 8600 GT (32 Núcleos CUDA)
Memoria principal: 2GB
Memoria Gráfica: 256 Mb
Sistema operativo: Debian 6
OpenCL: 1.1 (Cuda 4.0)