Comparativa del Algoritmo Quicksort

Arquitecturas Avanzadas de Computo

Bruno Baruffaldi

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

1. Objetivo:

El siguiente informe es un trabajo final correspondiente al curso Arquitecturas avanzadas de cómputo dictado durante la escuela de ciencias informáticas en la uba en el 2018, cuyo objetivo es realizar una comparación entre los tiempos de ejecución de un algoritmo de ordenamiento implementado sobre distintas arquitecturas y tratando de aprovechar al máximo la paralelización de tareas.

2. Introducción:

Existen un conjunto de distintas arquitecturas que fueron creadas para resolver tareas específicas, pero actualmente pueden llegar a ser útiles para la resolución de problemas de cualquier índole. Dichos problemas, dependiendo de la cantidad de operaciones que realizan, pueden llegar a tardar un tiempo prolongado para su resolución. Sin embargo, en algunos casos estos tiempos pueden ser ampliamente mejorados con la programación en paralelo. Existen diferentes formas de ejecutar programas en paralelo y la mayoría depende del hardware que se utiliza.
En un comienzo la programación en paralelo era simulada debido al alto costo del hardware, pero aun así se obtenían mejores resultados. Hoy en dia, en la mayoría de los dispositivos informáticos que utilizamos cotidianamente es habitual que posean un procesador multinúcleo, que combina dos o más microprocesadores en un solo circuito integrado, lo que permite que el programador cuente con una forma de paralelismo real a nivel de threads. Para facilitar al programador hacer uso de esta ventaja se desarrollaron distintas herramientas como OpenMP o POSIX Threads que permite aadir concurrencia a programas desarrollados en C,C++ y Fortran.
Otra arquitectura interesante son las unidades de procesamiento gráfico o GPU, que son dispositivos desarrollados para el procesamiento de gráficos u operaciones de punto flotante que cuentan con una gran cantidad de núcleos. Con el tiempo, estos dispositivos empezaron a ser utilizados para el cálculo científico de propósito general (en inglés GPGPU - General-Purpose Computing on Graphics Processing Units) dado que de esta forma muchas aplicaciones mejoraron enormemente sus tiempos de ejecución. Por estos motivos, empresas como NVIDIA introdujeron en sus tarjetas gráficas la arquitectura CUDA para el cálculo paralelo de propósito general.

3. Algoritmo:

En este informe se decidió utilizar el algoritmo de ordenamiento Quicksort y comparar sus tiempos de ejecución de distintas implementaciones que buscan explotar las ventajas de diferentes arquitecturas.
El algoritmo quicksort es uno de los algoritmos de ordenamiento más eficientes y funciona de la siguiente forma:

1. Elige un valor del arreglo al cual llamaremos pivote.
2. Resitua a los demás elementos de la lista a cada lado del pivote, insertando a todos los elementos menores que el pivote del lado izquierdo y a los mayores del lado derecho.
3. De esta forma, las lista inicial queda separada en dos sublistas, una con los elementos menores que el pivote y otra con los elementos mayores por lo cual pueden ser ordenadas de forma independientes. Notar que el pivote se encuentra en su posición correspondiente respecto del arreglo ordenado.
4. El algoritmo repite el proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado el proceso el arreglo está ordenado.

Este algoritmo fue creado por el científico británico Tony Hoare, quien sugirió que cuando la lista de números a ordenar no es demasiado pequea es mejor utilizar otro algoritmo.
La programación en paralelo consiste en dividir un problema en subproblemas (lo más independientes posibles) con el fin de optimizar el tiempo de ejecución del programa. Esto es exactamente lo que hace el algoritmo quicksort y es lo que se trató de explotar en las distintas implementaciones para reducir el tiempo de ejecución.

4. Herramientas Utilizadas:

4.1. Lenguaje C:

C es un lenguaje de propósito general desarrollado en 1970 por Dennis M. Ritchie. Desde su desarrollo C tuvo un rol central en UNIX y Linux, donde se destaca que aproximadamente el 97 % del kernel de Linux esté programado en C.
C presenta muchas ventajas que atraen a los programadores, entre ellas:

- Ofrece características de bajo nivel. Es posible programar en lenguaje ensamblador desde C y realizar modificaciones a nivel de bit.
- C es portátil, si no se utilizan las bibliotecas específicas de OS, C puede funcionar en cualquier computadora sin necesidad de modificaciones.
- Posee soporte para asignación de memoria dinámica y desasignación usando punteros.

4.2. Posix Threads:

Posix Threads, usualmente conocido como pthreads, es una librería que define un conjunto de funciones, tipos y constantes para facilitar el manejo de threads en C.
Un thread es un proceso que se ejecuta en el sistema operativo, este cuenta su propio número de proceso, su propia pila y una copia de los registros del procesador pero puede compartir a sus vez memoria o estructuras con otros procesos. Nuestro enfoque va a estar basado en la utilización de estos threads para mejorar el rendimiento del algoritmo.
Esta librería cuenta con cientos de herramientas para la utilización de semáforos de dijkstra, mutexes y el manejo de hilos.Sin embargo, es bastante robusta y requiere de bastantes conocimientos acerca de la programación multithreading para poder utilizarla.

4.2.1. OpenMP:

OpenMP es una librería para crear programas paralelos que permite utilizar memoria compartida que se compone de un conjunto de directivas de compilador o pragmas lo que permite que un programa que utiliza esta herramienta pueda ser compilado por compiladores que no soporten OpenMP.
Esta librería permite mucha flexibilidad en el código haciendo este tipo de implementaciones relativamente sencillas. Para crear los hilos cuando se encuentra una directiva paralela OpenMP usa un modelo llamado fork-join.
Esto significa es que cuando un thread de ejecución encuentra una construcción paralela creará un equipo de threads(fork) y se convertirá en el thread maestro del equipo. Luego, el equipo ejecuta el código asignado a él y se espera a que todos los threads terminen antes de que el thread maestro continúe con la ejecución del código que se encuentra después de la construcción en paralelo(join).

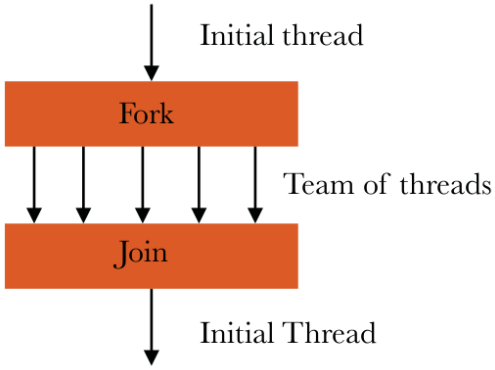

Figura 1: Modelo Fork-Join

4.3. CUDA:

Cuda es una tecnologia desarrollada por Nvidea que permite a los programadores ejecutar código C directamente en la GPU. Esto permite ejecutar cientos o miles de hilos simultáneamente.
En la arquitectura clásica de una tarjeta grafica podemos encontrar la presencia de dos tipos de procesadores, los procesadores de vértices y los de fragmentos, que cuentan con repertorios de instrucciones diferentes. Sin embargo en la arquitectura CUDA todos los núcleos comparten el mismo repertorio de instrucciones y prácticamente los mismos recursos. En esta arquitectura están presentes unas unida-

Figura 2: Arquitectura CUDA

des de ejecución denominadas Streaming Multiprocessors que están interconectadas entre sí por una zona de memoria común. Estas a su vez están compuestas por unos núcleos de cómputo llamados ñúcleos CUDAö Streaming Processors que son los encargados de ejecutar las

instrucciones. Este diseño permite permite la programación sencilla de los núcleos de la GPU utilizando un lenguaje de alto nivel como puede ser el lenguaje C. Para ello, el programador escribe un pro-

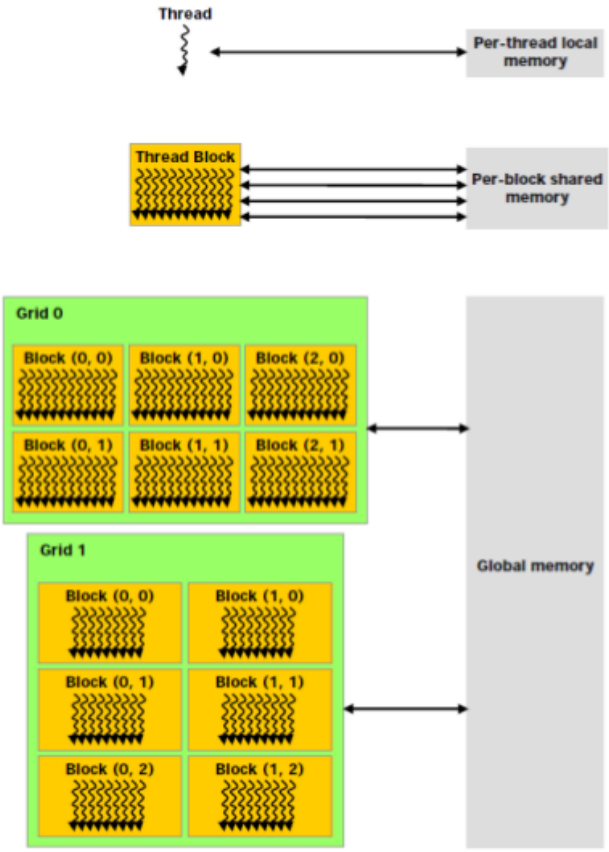

Figura 3: Jerarqua de hilos en una aplicacin CUDA.

grama secuencial que se conoce como kernel que se ejecuta dentro de la GPU como un conjunto de hilos y que se organizan dentro de una jerarquía en la que pueden agruparse en bloques, y que a su vez se pueden distribuir formando una malla (cuando se invoca un kernel, el programador especifica el número de hilos por bloque y el número de bloques que conforman la malla). En cuanto a la memoria, durante su ejecución los hilos pueden acceder a los datos desde diferentes espacios dentro de una jerarquía de memoria. Así, cada hilo tiene una zona privada de memoria local y cada bloque tiene una zona de memoria compartida visible por todos los hilos del mismo bloque, con un elevado ancho de banda y baja latencia. Finalmente, todos los hilos tienen acceso a un mismo espacio de memoria global ubicada en un chip externo de memoria DRAM.

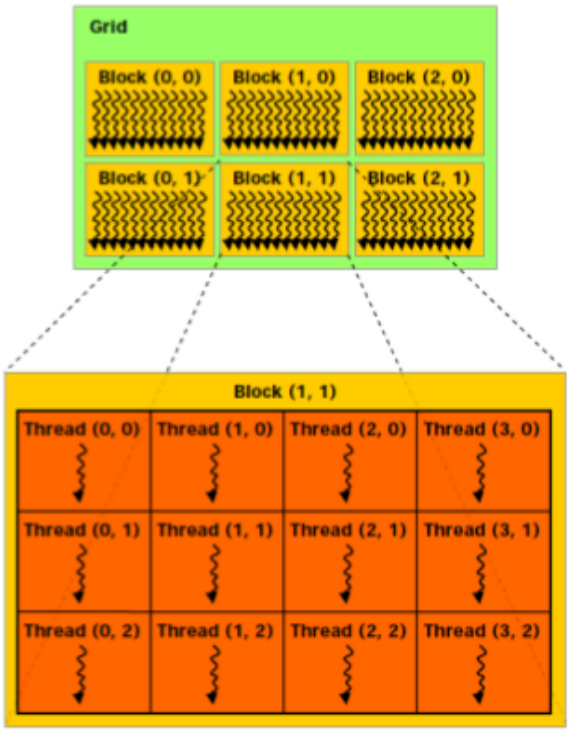

Figura 4: Jerarqua de memoria dentro de la arquitectura CUDA.

En este modelo de programación el programador debe copiar los datos desde la CPU al dispositivo para poder utilizarlos.

5. GPU-QuickSort:

La programación en paralelo consiste en dividir un problema en subproblemas(lo más independientes posibles), con el fin de optimizar el tiempo de ejecución del programa. Esto es exactamente lo que hace el algoritmo quicksort y es lo que se trató de explotar en las distintas implementaciones para reducir el tiempo de ejecución.
En las implementaciones de OpenMP y Posix fue relativamente sencillo paralelizar el código por la forma del algoritmo. Simplemente se debía paralelizar las llamadas recursivas de ser conveniente luego de ordenar el arreglo respecto de un pivote. Sin embargo, la implementación en CUDA no fue tan directa, ya que por su arquitectura utilizar un único thread para ordenar el arreglo parcialmente era muy costoso y ralentizaba mucho el algoritmo. Es por esto que se optó por una adaptación del algoritmo para GPU.

La esencia del algoritmo GPU-Quicksort sigue siendo la misma que la del algoritmo original, donde en cada instancia se ordena el arreglo respecto de un pivote generando dos nuevos arreglos a ser ordenados independientemente. Al principio la cantidad de subsecuencias a ordenar va a ser muy baja por lo que varios bloques van a tener que trabajar juntos para ordenar una misma secuencia de elementos. Esto requiere una sincronización apropiada entre los distintos hilos, pues los resultados deben ser combinados para obtener las dos subsecuencias resultantes.
La GPU modernas soportan un repertorio de primitivas que permiten hacer determinadas operaciones de manera atómica. Un ejemplo de esto es atomicAdd, que permite incrementar el valor de una variable compartida por varios hilos de forma atómica sin necesidad de utilizar una barrera para sincronizar todos los threads. Estas funciones fueron fundamentales para la implementación del algoritmo.
La razón de esto es que no hay forma de saber en qué orden se ejecutarán los bloques o si se ejecutan todos a la vez. Entonces, la única forma de sincronizar los hilos de distintos bloques es dividir el algoritmo en distintos kernels y lanzarlos secuencialmente, de esta forma nos aseguramos de que todos los bloques terminaron su ejecución hasta cierto punto. Salir y entrar de la GPU no es costoso, pero tampoco es inmediato ya que los parámetros deben ser copiados de la CPU a la GPU cada vez. Esto implica que debemos minimizar el número de veces que utilizamos este recurso.
Vale aclarar que no es conveniente utilizar la GPU cuando si la cantidad de elementos que posee una subsecuencia no es muy grande, ya que lleva más tiempo la comunicación con la GPU que ordenar el arreglo directamente desde la CPU.
Es por esto que se decidió organizar el algoritmo en dos fases.
En la primera fase el objetivo es dividir el arreglo en subsecuencias más pequeñas que posean a lo sumo una cantidad de elementos determinada (en nuestro caso es la constante MAXSEQ). Durante la segunda fase se procede a ordenar las subsecuencias obtenidas para finalizar de ordenar el arreglo.
Una de las ventajas del quicksort es que es un algoritmo in-place, lo que brinda un buen comportamiento de la caché del procesador en sistemas convencionales. Sin embargo, en la GPU nos encontramos con una memoria caché más limitada y una sincronización entre los thread más costosa. Es por esto que se utiliza un utiliza un buffer auxiliar con el fin de minimizar la comunicación entre los thread y no cargar tanto la caché de los núcleos cuda.
Una secuencia para ser ordena parcialmente es dividida desde la CPU en m subsecuencias (en lo posible de igual tamaño) a las que

luego se les asignará un bloque de threads para su ejecución en la GPU. El objetivo de esto es tratar de paralelizar la partición de un arreglo respecto de un pivote minimizando la sincronización entre los threads.

Para llevar a cabo la primera fase se le asigna un id a cada bloque y

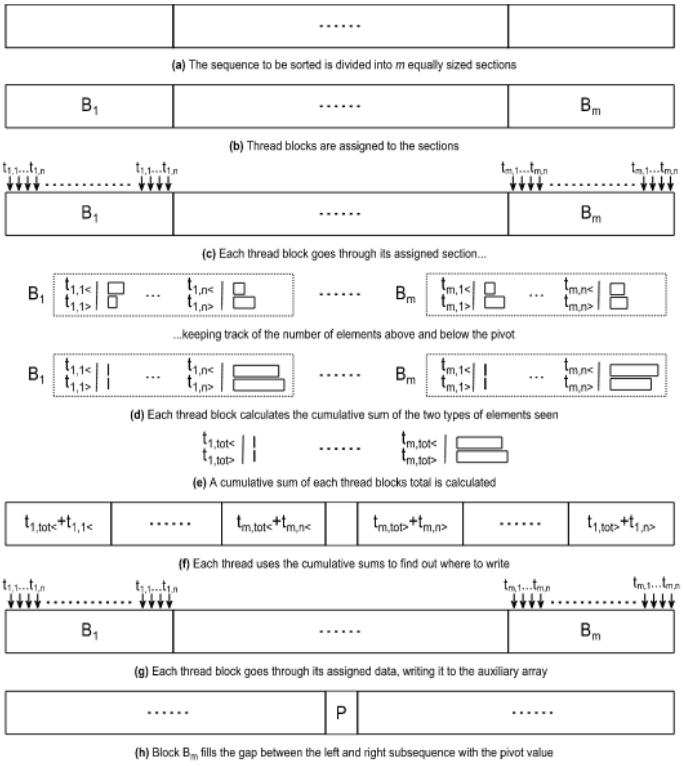

Figura 5: Particion de una secuencia respecto de un pivote.

se calcula cuántos elementos menores que el pivote se encuentran en la subsecuencia de cada bloque y se almacena en un arreglo auxiliar. Luego se calculan las sumas parciales para determinar a partir de qué posición se deben situar los valores menores que el pivote de la secuencia que estamos manejando en el buffer auxiliar donde se almacenarán el arreglo parcialmente ordenado. De esta forma, cada bloque sabe que x bloques con un id menor quieren escribir un total y elementos menores que el pivote en el buffer auxiliar. Por lo tanto, cada bloque puede empezar a escribir los valores menores que el pivote en el buffer auxiliar de partir de la posición y+1 de manera independiente con respecto a los demás bloques. Se procede de manera análoga con los valores mayores que el pivote y una vez hecho esto como paso final copiamos el valor del pivote en el espacio

que quedó entre las dos subsecuencias. Finalmente el pivote se encuentra ahora en su posición final con respecto al arreglo ordenado por lo que no debe ser incluido en las subsecuencias generadas. De esta forma logramos paralelizar el ordenamiento parcial del arreglo desde la GPU de una manera bastante eficiente.
Para implementar la primera fase fue necesario definir tres kernels que se ejecutarán secuencialmente, cadas uno de ellos con un propósito específico:

1. El primero se encarga de contar la cantidad de elementos mayores y menores que se encuentran en la secuencia asignada. Para ello cada thread del bloque recorre una parte de la secuencia lleva la cuenta de los elementos en variables locales, para luego sumar atómicamente estos valores en los arreglos auxiliares LT y GT.

2. El segundo se encarga de copiar los valores del arreglo a ordenar en un buffer auxiliar. Cada bloque calculará la suma correspondiente a su id independientemente de los demás para evitar la necesidad de sincronización entre ellos. Una vez hecho esto podemos determinar a partir de donde empezar a copiar los valores mayores y menores al pivote en el buffer auxiliar. Finalmente, el copia el valor del pivote en el espacio correspondiente y se calculan las nuevas subsecuencias.

3. El tercero se encarga simplemente de copiar los valores del buffer auxiliar al arreglo original para poder repetir el proceso si continuar ordenando el arreglo.

Entre cada iteración, se traen al CPU las nuevas secuencias a ordenar para dividirlas en subsecuencias a las cuales se les asignará un bloque de thread en la GPU y se descartaran las secuencias que no sean lo suficientemente grandes.
La segunda fase comienza cuando todas las secuencias son lo suficientemente cortas como para reiterar nuevamente en la GPU, por lo que se procede a ordenarlas de forma independiente.

6. Experimentos:

Los programas utilizados están disponibles en un repositorio de Github y fueron ejecutados en cluster de la Universidad Nacional de rosario que cuenta con un procesador Intel Xeon E5506 con 4 nucleos, 4 MB de cache y una frecuencia de 2.13 GHz y una tarjeta gráfica NVIDIA Tesla K40c con 2880 núcleos cuda y 12 GB de

memoria. Para obtener estos resultados se fueron modificando las macros de los programas para establecer el número de elementos a ordenar, la cantidad de hilos y otros valores para tratar de sacar el mayor provecho al algoritmo.
La siguiente tabla muestra una comparativa entre los tiempos de las distintas implementaciones en milisegundos respecto a la cantidad de elementos a ordenar.

Cantidad	Secuencial	OpenMp	Posix	Cuda
1,000	0.196	0.433	0.210	3.555
5,000	1.202	1.662	1.335	6.226
10,000	2.391	3.029	2.524	9.962
50,000	12.236	16.156	13.202	12.248
100,000	28.779	23.569	27.818	19.947
500,000	143.603	60.551	93.471	37.349
1,000,000	304.991	111.389	203.211	46.617
5,000,000	1,698.34	531.553	1,122.590	88.937
10,000,000	3,683.713	1,110.505	2,473.136	158.088
50,000,000	20,287.507	6,245.805	13,316.552	401.351
100,000,000	41,183.724	13,359.834	32,087.728	878.485
500,000,000	224,981.828	69,228.161	148,800.042	3,803.709
1,000,000,000	467,772.651	142,235.537	291,855.056	7,313.001

Tener en cuenta que los arreglos fueron creados con números pseudo-aleatorios y no sobre alguna distribución en particular.

7. Conclusión:

Como se observa en los experimentos cada implementación hace mejor o peor tiempo dependiendo del número de elementos que posea el arreglo.
Si la cantidad de elementos es pequea (menos de 100000), sería conveniente utilizar la versión secuencial del algoritmo sin ninguna optimización en cuanto a la paralelización pues el algoritmo corre lo suficientemente rápido y no pierde tiempo en la creación de nuevos hilos.
Si la cantidad de elementos pasa a ser un poco más grande (entre 500000 y 5000000) es útil utilizar varios núcleos del procesador para ejecutar varios hilos y paralelizar la operaciones.
Sin embargo, a si la cantidad de números es muy grande la mejor opción es la implementación GPU del algoritmo. Esta última versión es varias veces mas rapida que la anterior, si bien en un principio

se puede pensar que esto es debido a la superioridad del dispositivo GPU con respecto a la CPU sobre la cual se llevaron a cabo los experimentos. No obstante, si bien esto puede llegar a influir en los experimentos no se puede negar la amplia mejora en los tiempo de ejecución del algoritmo.