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 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 añadir 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 los 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 grande es mejor utilizar otro algoritmo.

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 algún sistema operativo, 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.
Esta librería cuenta con cientos de herramientas para la utilización de semáforos de dijkstra, mutexes y el manejo de threads.Sin embargo, es bastante robusta y requiere de bastantes conocimientos acerca de la programación multithreading para poder utilizarla.

4.3. OpenMP:

OpenMP es una librería para crear programas paralelos que permite utilizar memoria compartida y 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.

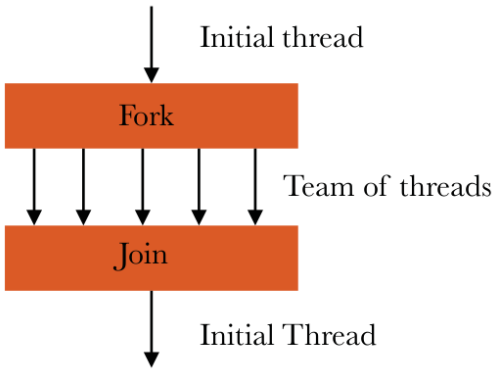

Figura 1: Modelo Fork-Join

Para crear los threads 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 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).

4.4. CUDA:

Cuda es una tecnologia desarrollada por Nvidea que permite a los programadores correr código C directamente en la GPU. Esto permite ejecutar cientos o miles de threads 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.

Figura 2: Arquitectura CUDA

En esta arquitectura están presentes unas unidades 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 núcleos CUDA o Streaming Processors que son los encargados de ejecutar las instrucciones. Este diseño 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 programa secuen-

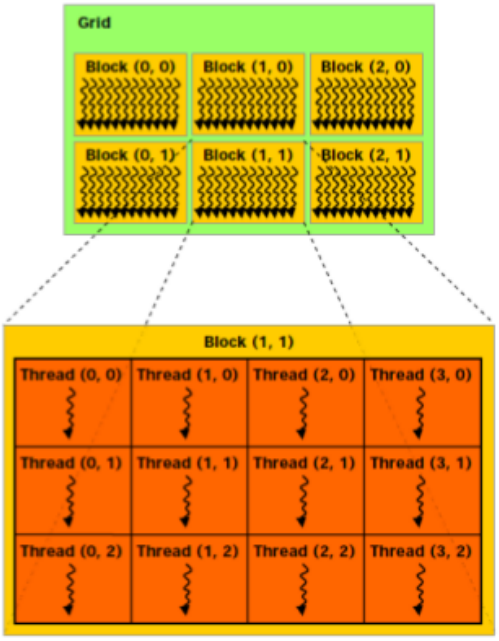

Figura 3: Jerarqua de threads en una aplicacin CUDA.

cial que se conoce como kernel que se ejecuta dentro de la GPU como un conjunto de threads. Estos threads se organizan dentro de una jerarquía en la que pueden agruparse en bloques, los que a su vez se distribuyen formando una malla (cuando se invoca un kernel, el programador especifica el número de threads por bloque y el número de bloques que conforman la malla).

En cuanto a la memoria, durante su ejecución los threads pueden acceder a los datos desde diferentes espacios dentro de una jerarquía de memoria. Así, cada thread tiene una zona privada de memoria local y cada bloque tiene una zona de memoria compartida visible por todos los threads del mismo bloque, con un elevado ancho de banda y baja latencia. Finalmente, todos los threads tienen acceso a un mismo espacio de memoria global ubicada en un chip externo

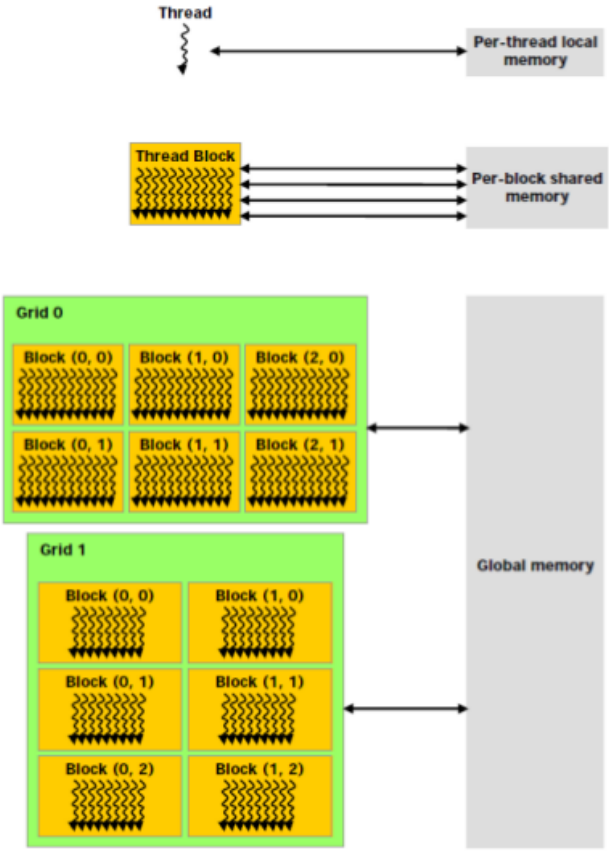

Figura 4: Jerarqua de memoria dentro de la arquitectura CUDA.

de memoria DRAM.
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 imple-

mentación 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 threads, 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 threads 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 threads 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 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. 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.

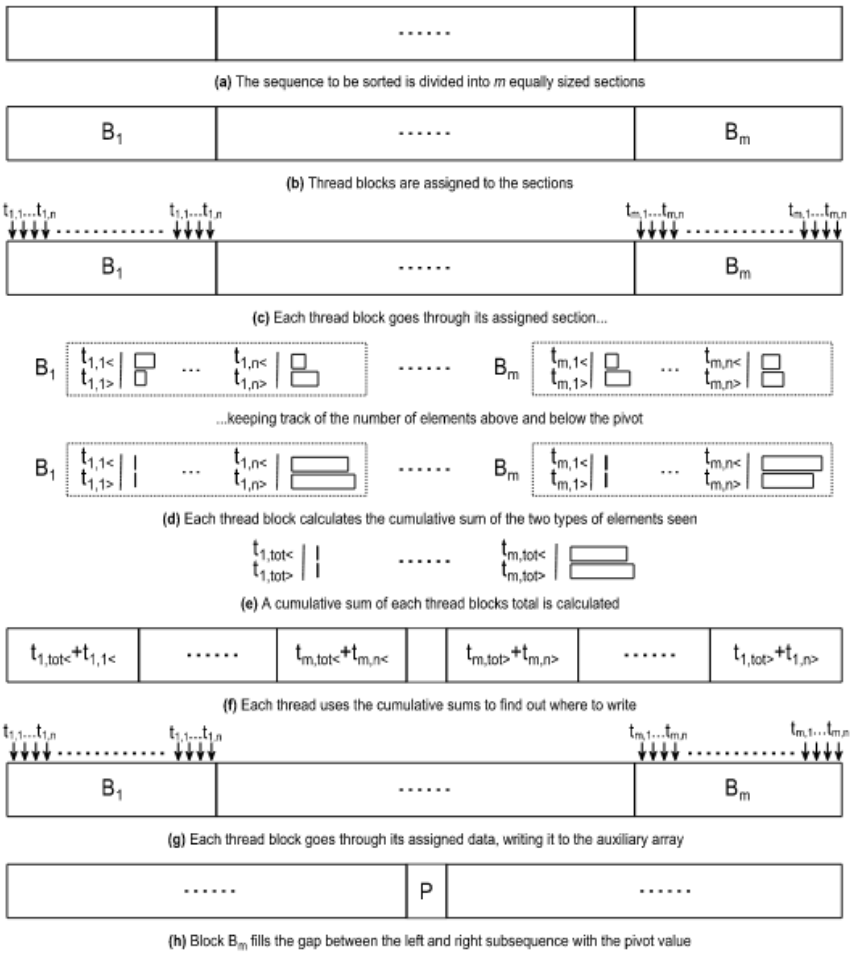

Figura 5: Particion de una secuencia respecto de un pivote.

Para llevar a cabo la primera fase se le asigna un *id* a cada bloque, se calcula cuántos elementos menores que el pivote se encuentran en la subsecuencia correspondiente a 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 almacenará el arreglo parcialmente ordenado. De esta forma, cada bloque sabe que x bloques con un id menor quieren escribir un total de 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 a partir de la posición $y+1$ de forma 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 el pivote que se encuentran en la secuencia asignada. Para ello cada thread del bloque recorre una parte de la secuencia llevando una cuenta de los elementos en variables locales, para luego sumar atómicamente estos valores desde los arreglos auxiliares.

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, se 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 y 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 un 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.
La siguiente tabla muestra una comparativa entre los tiempos de ejecucion de las distintas implementaciones en milisegundos respecto a la cantidad de elementos a ordenar.

Cantidad	Secuencial	Posix	OpenMp	Cuda
1,000	0.089	0.089	0.309	2.086
5,000	0.492	0.490	0.810	2.381
10,000	0.967	0.957	1.409	2.391
50,000	4.914	4.480	6.374	4.592
100,000	10.475	9.152	12.150	12.947
500,000	38.939	26.777	20.649	24.399
1,000,000	93.105	66.961	40.727	25.257
5,000,000	320.312	168.560	109.947	59.336
10,000,000	919.456	602.068	314.666	95.199
50,000,000	2,305.785	1,373.217	763.572	362.211
100,000,000	8,642.571	5,501.847	2,905.469	695.845
500,000,000	17,949.616	10,816.509	5,825.085	3,243.294
1,000,000,000	89,075.281	52,898.419	28,090.016	6,533.099

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, existe una clara ventaja en la paralelización a nivel de threads y esto se ve reflejado en la reducción en los tiempos de ejecución de las diferentes implementaciones. Si se realiza la comparación entre la versión secuencial y la que utiliza las herramientas POSIX o OpenMP, se puede deducir que estas últimas reducen los tiempos de ejecución a casi la mitad.Es por esto que, si bien paralelizar un programa en la mayoría de los casos exige un esfuerzo extra, utilizar este tipo de librerías casi siempre será provechoso para el programador.
Asimismo, podemos notar que el programa que incluye OpenMP

mejora los tiempos de POSIX. Por esta razón y por ser mucho más flexible, podría considerarse a OpenMP como una mejor opción a la hora de llevar a cabo este tipo de implementaciones. Aunque también es cierto que POSIX presenta una gran cantidad de utilidades y permite hacer configuraciones más específicas a nivel de código.
Finalmente, es claro que si la cantidad de elementos a ordenar es muy grande nuestra mejor opción es recurrir a la implementación que involucra a la GPU. Sin embargo, el código de esta última implementación es mucho más complejo y requiere de mayores conocimientos y esfuerzo que las implementaciones anteriores. Podemos observar además que si la cantidad de números a ordenar es muy pequeña, los tiempos de ejecución estarán dominados en mayor medida por la comunicación entre la CPU y la GPU.

8. Posibles extensiones:

Una de las principales mejoras que se le pueden aplicar a este trabajo es comparar los tiempos de las distintas implementaciones sobre varias distribuciones diferentes para tener una mejor comprensión del algoritmo y poder determinar cuales son los puntos débiles de cada implementación. Además, esto nos permitirá tener una visión más completa de los tiempos de ejecución y la comparación entre los programas.
Otro gran punto a tratar son las optimizaciones en el código del algoritmo GPU-Quicksort. Por ser esta una de mis primeras experiencias en programación con este tipo de arquitecturas y no tener conocimientos muy profundos sobre la herramienta Cuda, creo que en el código pueden efectuarse grandes optimizaciones para explotar aún más las ventajas que nos brinda la GPU para reducir los tiempos de ejecución.
Otra posible extensión de este trabajo podría ser la implementación del algoritmo en OpenCL para así poder comparar dicha herramienta con Cuda y tener un mejor entendimiento acerca de la computación de alto rendimiento y de cómo explotar al máximo los distintos recursos que tengamos a disposición.