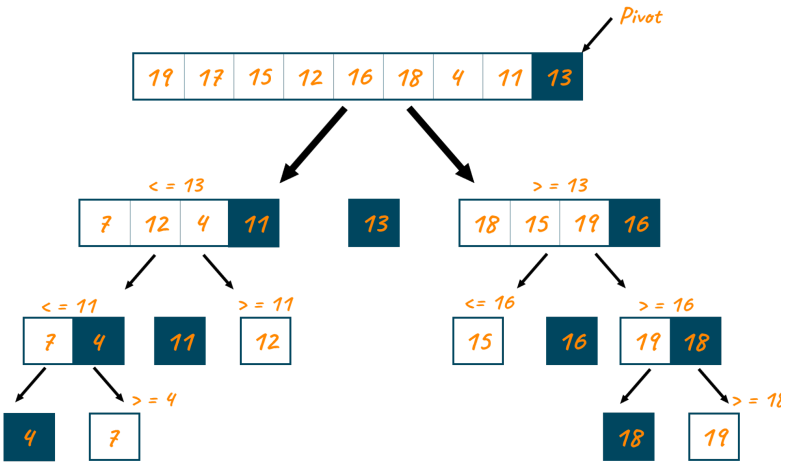
Proyecto TGA

Análisis del Quicksort en CUDA



Eduardo Gimeno Borrás

Carlos Fulgencio Velasco

Q1 2023-24

Índice

1. Propuesta de trabajo

1.1. Explicación algoritmo de la propuesta

Finalmente, la elección de nuestra propuesta de trabajo, después de haber estado barajando diversas opciones, es el análisis del algoritmo de ordenación *Quicksort*.
Nos decantamos por esta opción debido a que teníamos un conocimiento previo del algoritmo al haberlo trabajado en asignaturas anteriores.

Este algoritmo de ordenación se basa en la técnica de “divide y vencerás” por la que en cada recursión, el problema se divide en subproblemas de menor tamaño y se resuelven por separado (aplicando la misma técnica) para ser unidos de nuevo una vez resueltos.
Actualmente, es el método más eficiente y rápido de ordenación al realizar las menos operaciones posibles con un tiempo de ejecución promedio de O(n log (n)), siendo en el peor de los casos $O(n^2)$.

Para poder trabajar con las GPUs de Nvidia utilizamos CUDA, esta herramienta es una plataforma de computación paralela creada por NVIDIA que permite a los desarrolladores aprovechar el poder de cálculo de sus tarjetas gráficas. La programación en paralelo que nos ofrece CUDA es un enfoque importante para resolver problemas con alta necesidad de cálculo y optimización.

Previo a la explicación del código implementado en CUDA, detallaremos el algoritmo de *Quicksort* mediante el código secuencial. En la primera llamada a la función se envía todo el vector, su inicio y su final. A continuación, llama a una función partition que recibe el array y los índices y lo que retorna es el índice más a la izquierda, donde ya no ha encontrado elementos menores que un pivote, por el cual intercambiar por un elemento mayor que el pivote. Por lo tanto, se realizarán otras dos llamadas a la función para que evalúen una subparte del vector probando otros pivotes. La ordenación acabará cuando se haya ordenado todo el vector, o lo que es lo mismo, el índice de la derecha haya visitado todos los elementos hasta el final del vector.

```
C/C++
quicksortSeq(array, 0, N-1);

void quicksortSeq(int *array, int left, int rigth){
       if (left < rigth){
```

```
            int pi = partition(array,left,rigth);

            quicksortSeq(array,left,pi);
            quicksortSeq(array,pi+1,rigth);
        }
}
```

Una vez explicado el algoritmo que hemos implementado, nos falta comentar en detalle la función de partition. La función selecciona como pivote el primer elemento del vector, es decir, el de la izquierda del todo y a partir de ahí busca intercambiar un elemento menor que el pivote que se encuentre a la izquierda por un elemento mayor que el pivote que se encuentre a la derecha. En el caso de que haya evaluado todos los elementos de la derecha, el índice j llega a un punto donde es menor o igual que el índice i, sabremos que el vector ya no tiene más elementos que intercambiar y acaba devolviendo el índice que indica hasta donde está ordenado.

```C/C++
void swap(int *xp, int *yp){
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}

int partition(int *array, int left, int right){
  int x = array[left];

  //Los primeros valores no dan segmentation fault porque se ejecuta el do
antes de acceder al vector
  int i = left-1;
  int j = right+1;

  for(;;){
    do {
      i++;
    }while(array[i] < x);

    do{
      j--;
    }while(array[j] > x);
    //Cuando ya no encuentra más grandes a la izquierda del pivote, acaba
    if(i >= j) return j;
    swap(&array[i],&array[j]);
  }}
```

1.2. Referencias código secuencial

Como hemos comentado anteriormente, este algoritmo lo trabajamos en una asignatura del grado, por lo tanto, las referencias y el código se encuentran en los apuntes de la asignatura llamada EDA. Concretamente en los apuntes del tema *divide y vencerás*.

https://www.cs.upc.edu/eda/data/uploads/eda-codis.pdf

2. Implementación en CUDA

En CUDA la idea del algoritmo es la misma, elegir un pivote y mover a la izquierda los menores que el pivote y a la derecha los mayores. Lo que cambia ahora es que queremos realizar esta ordenación paralelamente, en vez de esperar a que la recursividad de un nivel inferior acabe. Con otras palabras, explotar lo máximo posible los threads disponibles para hacer el *Quicksort* lo más eficiente y rápido posible.

Para realizar esta implementación nos basamos en un paper llamado *GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors*, pero al irlo implementando y ver los problemas y soluciones que encontrábamos, vimos que era demasiado complejo para poder hacer la versión completa. Por este motivo, nos centramos en ordenar todo el vector, lo que hace es mover a la izquierda los menores que el pivote y a la derecha los mayores, esto permite que la CPU tan solo tenga que ordenar fracciones más pequeñas del vector.

Un gran cambio respecto al código secuencial es que se añade un vector contador auxiliar. Este vector contará cuántos elementos hay mayores y menores que el pivote en su bloque para posteriormente indexarlo al vector. El conteo del bloque se reparte entre los threads, en el cual cada thread realiza este contador en su parte asignada y se realiza la suma del bloque, esto permite que cada thread sepa el inicio y el fin de la escritura.

En primer lugar, inicializamos todas las variables que necesitamos y repartimos los bloques, para posteriormente realizar el conteo de cada thread.

Las variables shared son variables que comparten todo el bloque, los vectores lt y gt contienen nThreads valores para que cada thread pueda contar independientemente cuántos menores hay que el pivote y asignarlo en su región de lt, lo mismo para gt pero con los mayores. Las dos últimas variables son los índices donde empieza el bloque a escribir en el vector final.
Al principio ponemos el contador a 0 para evitar que haya problemas, a continuación dividimos el tamaño (blockT) del vector según cuántos bloques y threads hay para repartirlo equitativamente. Cuando ya tenemos el tamaño asignado, calculamos los índices de la parte del vector que va a tratar el thread, para realizar el conteo de mayores y menores.
Los valores que son iguales al vector no se tienen en cuenta, lo veremos más adelante.
Por último, sincronizamos los threads porque no podremos realizar la suma del bloque sin tener el valor de todos.

```C/C++
__shared__ int lt[nThreads];
__shared__ int gt[nThreads];
__shared__ int lfrom;
__shared__ int gfrom;


int idThread = threadIdx.x;
lt[idThread] = 0;
gt[idThread] = 0;

//gridDim es cuantos bloques hay
//blockDim es cuantos threads hay
int blockT = (end-start+(gridDim.x*blockDim.x-1))/(gridDim.x*blockDim.x);
int i_start = start + (blockIdx.x * blockDim.x * blockT) + threadIdx.x * blockT;
int i_end = i_start + blockT;
if ((blockIdx.x==gridDim.x-1) && (threadIdx.x==blockDim.x-1)) i_end=end;

//Cada thread cuenta cuantos mayores y menos al bloque asociado
for (int i=i_start; i<i_end; i++){
  if (array[i] > pivot) gt[idThread] = gt[idThread]+1;
  if (array[i] < pivot) lt[idThread] = lt[idThread]+1;
}

__syncthreads();
```

El thread 0 de cada bloque es el encargado de realizar la suma de los menores y mayores del bloque. La suma se guarda en un vector global, se recibe por parámetro para que todos los bloques pueda acceder de nBlocks elementos. Esta suma indica el número de elementos que el bloque escribirá en el vector final, para que posteriormente se calcule el índice del bloque a través de la suma acumulada de los bloques anteriores. Por ejemplo, el bloque 0 ya sabemos que escribirá al inicio los menores y al final los mayores, el bloque 1 escribirá a continuación del número de elementos menores que el pivote que haya encontrado el bloque 0 y así sucesivamente.

Debido a esta suma acumulada, cada thread ya sabrá donde escribir sin crear problemas de data race por sobreescribir en la misma zona y así se podrán realizar paralelamente los accesos.

```C/C++
if (threadIdx.x == 0){
  int ltsum = 0;
```

```
  int gtsum = 0;
  for(int k = 0; k < nThreads; k++){
    ltsum += lt[k];
    gtsum += gt[k];
  }

  // Se guarda el valor de la suma para saber donde se empezará a indexar el
bloque
  int idblock = blockIdx.x;
  lbeg[idblock] = ltsum;
  gbeg[idblock] = gtsum;

  // Se calculan los indices de donde empieza el bloque
  lfrom = 0;
  gfrom = 0;
  for(int j = 0; j < blockIdx.x; ++j){
    lfrom += lbeg[j];
    gfrom += gbeg[j];
  }
}
```

Una vez ya tenemos los índices del bloque hay que calcular los respectivos índices de cada thread y escribir en el vector. Los respectivos idlt y idgt son los índices del arraysorted. El cálculo es el índice de bloque más el número de menores que ha encontrado el thread en el bloque asignado, por lo tanto, cada thread colocará en el vector final en un bloque diferente, ya que los menores que encuentra un thread son los que encuentra más sus anteriores. Para los mayores la idea es la misma, solo que ahora la inserción se realiza al final del bloque, para que los mayores queden a la derecha del pivote.

```C/C++
int idlt = lfrom + lt[threadIdx.x];
int idgt = end-1 - gfrom - gt[threadIdx.x];

//Se asignan los valores del vector original al vector que va a tener los
mayores a la derecha y los menores a la izquierda
for (int i=i_start; i<i_end; i++){
  if(array[i] < pivot) {
    arraysorted[idlt] = array[i];
    ++idlt;
  }
  if (array[i] > pivot) {
    arraysorted[idgt] = array[i];
    --idgt;
```

```
  }
}

__syncthreads();
```

Por último, hay que escribir el pivote o los pivotes en la región restante, es decir, al final del total de menores que han encontrado todos los threads de todos los bloques hasta el número de mayores encontrados menos el tamaño del bloque.

Creemos que puede haber un error en la inserción de los pivotes, ya que como veremos más adelante, cuando hay un número grande de elementos y un número pequeño de threads los pivotes no aparecen en el vector final. Por falta de tiempo y gran complejidad del problema inicial, no hemos podido resolver este detalle. Sin embargo, para tamaños pequeños el algoritmo funciona y esto también nos ha generado problema de saber dónde está el fallo.

```C/C++
if ((threadIdx.x == 0) && (blockIdx.x == 0)) {
  int ltsumt = 0;
  int gtsumt = 0;
  for(int n = 0; n < gridDim.x; n++){
    ltsumt += lbeg[n];
    gtsumt += gbeg[n];
  }

  int endp = end - gtsumt;
  for (int j = ltsumt; j < endp; j++){
    arraysorted[j] = pivot;
  }
}
```

2.1. Referencias código del kernel

Por suerte o por desgracia no hemos encontrado este código en CUDA en ningún sitio. Lo que sigue que conseguimos encontrar fue paper con una implementación compleja en pseudocódigo. El paper pertenece a Daniel Cederman and Philippas Tsigas y se llama "*GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors*".

Como hemos comentado en el punto anterior, hemos intentado entender e implementar el pseudocódigo del paper, pero nos ha resultado muy complejo. Para resolver nuestras dudas y que la implementación fuese posible, nuestro profesor de laboratorio (Daniel Jiménez González) nos comentó que implementar el índice mediante el conteo de mayores y menores sería mejor.

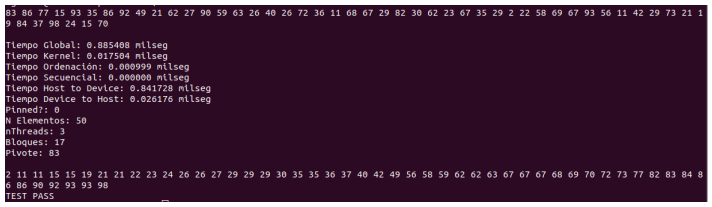
GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors:
https://www.cse.chalmers.se/~tsigas/papers/GPU-Quicksort-jea.pdf

3. Análisis de rendimiento

A continuación, analizaremos varios outputs del nuestro programa ejecutado en boada, en la salida podemos observar el tamaño de la entrada, tiempos de diferentes operaciones, número de threads y si es memoria pinned.

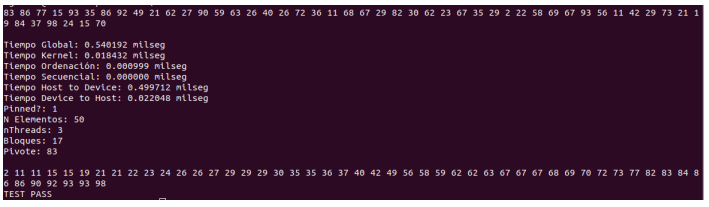
Hemos probado a aumentar el tamaño del vector y comprobar si el hecho que el número de threads sea múltiplo del tamaño del vector sigue funcionando.

No pinned

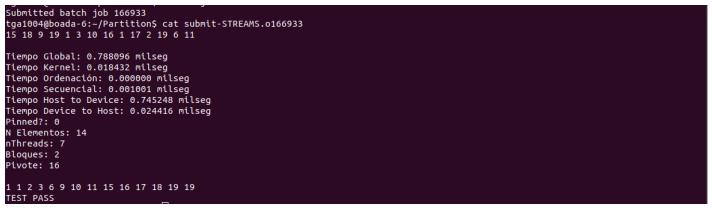


Ejemplo 1.1: Número de elementos 50 y nThreads 3.
No usamos memoria pinned y el tiempo total es de 0.885 milisegundos.

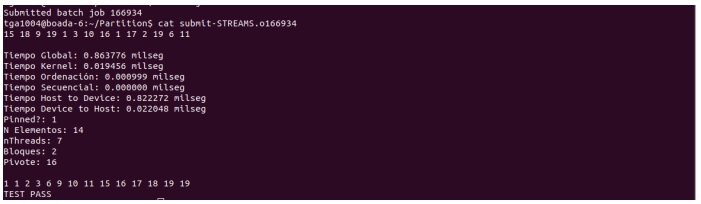
Pinned



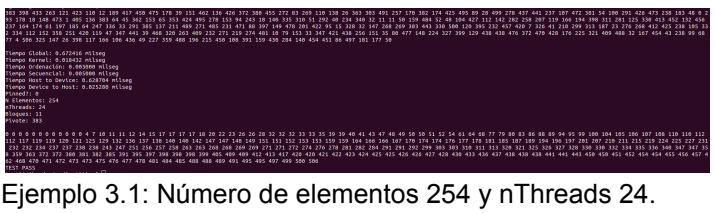
Ejemplo 1.2: Número de elementos 50 y nThreads 3.
Usamos memoria pinned y el tiempo total es de 0.540 milisegundos.



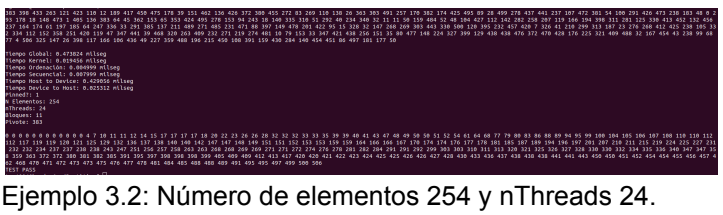
Ejemplo 2.1: Número de elementos 14 y nThreads 7.
No usamos memoria pinned y el tiempo total es de 0.788 milisegundos.



Ejemplo 2.2: Número de elementos 14 y nThreads 7.
Usamos memoria pinned y el tiempo total es de 0.863 milisegundos.



Ejemplo 3.1: Número de elementos 254 y nThreads 24.
No usamos memoria pinned y el tiempo total es de 0.672 milisegundos.



Ejemplo 3.2: Número de elementos 254 y nThreads 24.
Usamos memoria pinned y el tiempo total es de 0.473 milisegundos.

En el primer ejemplo podemos comprobar una diferencia notoria en el tiempo global del programa. Esto es algo que nos esperábamos debido a que estamos trabajando con un vector y el hecho de tener los elementos en direcciones de memoria en la misma página hace que los accesos sean más rápidos. En otras palabras, cuando accedemos a array[i] y seguidamente a array[i+1] al no tener que cargar otra página a memoria es más rápido. Además, en el segundo ejemplo, cuando hay pocos elementos, la diferencia no es tan elevada.

También observamos en los primeros ejemplos que independientemente de si los threads son múltiplos del tamaño del vector, el algoritmo lo ordena correctamente. Sin embargo, cuando el vector es de gran tamaño, no obtenemos la salida esperada. Creemos que este error está relacionado con la inserción del pivote debido a que los ceros del inicio son los que corresponden al número de pivotes y el pivote no se encuentra en el vector final. Como hemos comentado anteriormente, no hemos conseguido solventar el problema por temas de complejidad del algoritmo y por falta de tiempo.