Bitonic Sort

Alejandro Martínez Zamorano
Ricard Meyerhofer Parra
TGA 2015-2016Q2
Profesor: Agustín Fernández Jiménez

Índice

1. Introducción

1.1. Objetivos

En este trabajo se pretenden los siguientes objetivos:

- Creación de un código CUDA que use una y varias GPUs.
- Ejemplificar el proceso de paralelización de un algoritmo y el uso de múltiples GPUs mediante CUDA.
- Evidenciar el potencial de cálculo de varias GPUs en paralelo.

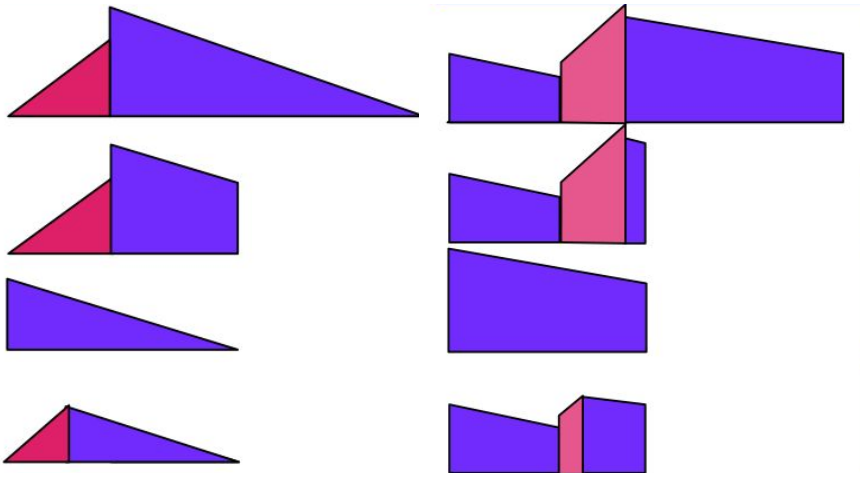
1.2. Propiedades Bitónicas

Para poder explicar el problema con mayor claridad se introducen las siguientes definiciones y propiedades:

- Una secuencia bitónica de aquella que crece sucesivamente hasta cierto punto donde decrece sucesivamente es decir $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ donde $0 \leq k < n$.
- Una secuencia ordenada es también una secuencia bitónica pero con una de sus partes vacía.
- Se consideran también secuencias bitónicas aquellas que son rotaciones cíclicas de un vector bitónico es decir por ejemplo: **<8,9,2,1,0,4>** es bitónico pues es una rotación cíclica de **<0,4,8,9,2,1>**.

Es conveniente notar que dos números son una secuencia bitónica pues estos pueden ser partidos en secuencias bitónicas más pequeñas (el elemento y vacío).

A modo ilustrativo se adjuntan algunas imágenes para ver diversas secuencias que son bitónicas (no representa el conjunto total de posibilidades):



1.3. Explicación del algoritmo

El algoritmo que hemos decidido implementar se trata del denominado Bitonic Sort. Éste se trata de un algoritmo de ordenación de números **naturales** por comparación, basado en convertir una secuencia desordenada en una bitónica. El algoritmo se compone de dos partes:

- Se construye una secuencia bitónica a partir de la secuencia desordenada (Bitonic build).
- Una vez aplicado el bitonic build, se divide la secuencia múltiples veces y se intercambian elementos hasta que se obtiene la secuencia ordenada (Bitonic merge).

Este algoritmo tiene un coste de $\Theta(n \log^2 n)$ en tiempo, lo cual lo aleja del coste "ideal" de un algoritmo de ordenación pero es puramente paralelizable, por lo tanto puede llevar a un mayor rendimiento al aplicar varias GPUs.

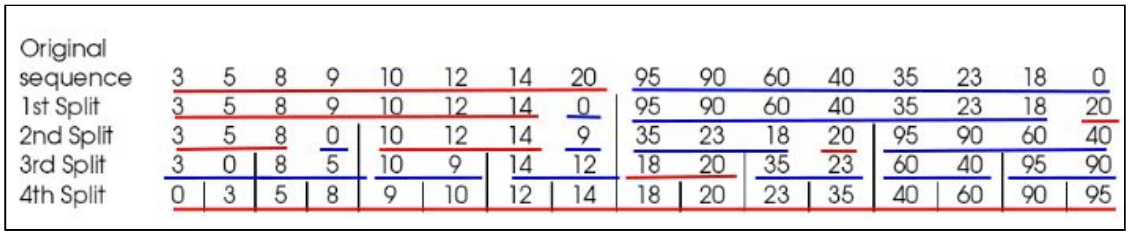
1.3.1. Bitonic Build

Nuestro objetivo es crear una secuencia bitónica. Para ello lo que hacemos es aplicar Bitonic Build a cada parte.

Esto nos generará una secuencia bitónica pues como anteriormente hemos comentado, llegará un punto donde haremos Bitonic Build de un vector de tamaño 2 lo cual nos permitirá ordenar correctamente y así sucesivamente.

1.3.2. Bitonic Merge

Este paso consiste en dividir el vector v en dos partes v_1, v_2 y conseguir que todos los elementos de una parte sean menores/mayores que los de la otra. Esto se puede conseguir si partimos en dos el vector v y comparamos para la posición relativa de v_1 a la de v_2 y en caso de que ésta sea mayor en v_2 que en v_1, entonces realizamos un intercambio de los valores de las posiciones. Si aplicamos recursivamente esto a cada parte, nos produce una secuencia ordenada.



1.4. Código C++

Bitonic.cpp

```
#include <iostream>
#include <vector>
using namespace std;

void bitonic_sort (vector<int> &v);
void bitonic_build (int l, int n, bool dir, vector<int> &v);
void bitonic_merge ( int l, int n, bool dir, vector<int> &v);
int greatestPowerOfTwoLessThan (int n);
void compare (int i, int j, bool dir, vector<int> &v);


int main() {
        int n;
        cin >> n;

        vector<int> v(n);

        for (int &i : v) cin >> i;

        bitonic_sort(v);
        for(int i : v) cout << i << " ";
        cout << endl;

}


void bitonic_sort (vector<int> &v) {
        bool ascending = true;
        bitonic_build(0, v.size(), ascending, v);
}

void bitonic_build (int l, int n, bool dir, vector<int> &v) {
        if (n > 1) {
           int m = n / 2;
           bitonic_build(l, m, !dir, v);
           bitonic_build(l + m, n - m, dir, v);
           bitonic_merge(l, n, dir, v);
        }
}
```

```
void bitonic_merge ( int l, int n, bool dir, vector<int> &v) {
        if (n > 1) {
            int m = greatestPowerOfTwoLessThan(n);
            for (int i = l; i < l + n - m; ++i) {
                compare(i, i+m, dir, v);
                bitonic_merge(l, m, dir, v);
                bitonic_merge(l + m, n - m, dir, v);
            }
        }
}

int greatestPowerOfTwoLessThan (int n) {
        int k = 1;
        while (k < n) k = k << 1;
        return k >> 1;
}

void compare (int i, int j, bool dir, vector<int> &v) {
        if(dir == (v[i] > v[j])) swap(v[i], v[j]);
}
```

1.5. Mejora Bitonic Sort

El bitonic sort es perfecto para la paralelización pues las llamadas recursivas se podrían hacer en paralelo y las comparaciones y swaps de elementos también.
Aún así es mejorable, pues si pasamos el código recursivo a uno iterativo lo expresaremos en términos de cada elemento (granularidad más fina) y podremos ejecutar en paralelo las siguientes operaciones:

- Todos los elementos ordenados son subsecuencias de un elemento.
- Parejas de elementos son subsecuencias ordenadas ascendente o descendentemente.

Bitonic_iterative.cpp

```
#include <iostream>
#include <vector>
using namespace std;

void bitonic_sort(vector<int> &v);
int greatestPowerOfTwoLessThan (int n);


int main() {
        int n;
        cin >> n;
```

```
        int m = greatestPowerOfTwoLessThan(n);
        vector<int> v(m);
        for (int i = 0; i < n; ++i ) {
            cin >> v[i];
        }
        for (int i = n; i < m; ++i) v[i] = 0x7FFFFFFF;
        bitonic_sort(v);
        for(int i : v) cout << i << " ";
        cout << endl;
}

void bitonic_sort(vector<int> &v) {
        for (int k = 2; k <= v.size(); k = 2 * k) {
            for (int j = k >> 1; j > 0; j = j >> 1) {
                for (int i = 0; i < v.size(); ++i) {
                    int ixj = i ^ j;
                    if ((ixj) > i) {
                        if ((i & k) == 0 and v[i] > v[ixj]) swap(v[i], v[ixj]);
                        if ((i & k) != 0 and v[i] < v[ixj]) swap(v[i], v[ixj]);
                    }
                }
            }
        }
}

int greatestPowerOfTwoLessThan (int n) {
        int k = 1;
        while (k < n) k = k << 1;
        return k;
}
```

Pensando ya en CUDA traducimos el código de C++ a C para usarlo en las siguientes versiones:

Bitonic_c.c

```
#include <stdio.h>

void swap (int i, int j, int v[]) {
    int aux = v[j];
    v[j] = v[i];
    v[i] = aux;
}

void bitonic_sort(int v[], int n) {
```

```
    int k, j, i;
    for (k = 2; k <= n; k = 2 * k) {
        for (j = k >> 1; j > 0; j = j >> 1) {
            for (i = 0; i < n; ++i) {
                int ixj = i ^ j;
                if ((ixj) > i) {
                    if ((i & k) == 0 && v[i] > v[ixj]) swap(i, ixj, v);
                    if ((i & k) != 0 && v[i] < v[ixj]) swap(i, ixj, v);
                }
            }
        }
    }
}

int greatestPowerOfTwoLessThan (int n) {
    int k = 1;
    while (k < n) k = k << 1;
    return k;
}


int main() {
    int n;
    scanf("%d", &n);

    int m = greatestPowerOfTwoLessThan(n);

    int v[m];
    int x, i;
    for (i = 0; i < n; ++i ) {
        scanf("%d", &x);
        v[i] = x;
    }
    for (i = n; i < m; ++i)
        v[i] = 0x7FFFFFFF;


    bitonic_sort(v, m);

    for (i = 0; i < m; ++i) {
        printf("%d ", v[i]);
    }
    printf("\n");
}
```

2. Bitonic Sort con una GPU

2.1. Implementación y explicación

Una vez tenemos el código en C y ya lo hemos optimizado de forma que sea altamente paralelizable, lo extenderemos a CUDA para que sea ejecutable en una GPU. Por el momento en una sola GPU pues la compartición de datos entre GPU's, no es del todo trivial y va a requerir de modificaciones en el código adicionales.

En este caso para ver que el código es correcto, hemos incorporado un bucle for que compruebe que las posiciones estén ordenadas ascendentemente (además de probar en casos pequeños para ver que funciona).

2.2. Código

unaGPU.cu

```
__global__ void bitonic_sort_step(int *dev_values, int j, int k){
  int i, ixj; // Sorting partners: i and ixj
  i = threadIdx.x + blockDim.x * blockIdx.x;

  ixj = i^j;
  if ((ixj) > i) {
    if ((i & k) == 0) {
      if (dev_values[i] > dev_values[ixj]) {
        int temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
    if ((i & k) != 0) {
      if (dev_values[i] < dev_values[ixj]) {
        int temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
  }
}
void bitonic_sort(int *dev_values){

  dim3 numBlocks(NUM_BLOCKS, 1);
  dim3 numThreads(NUM_THREADS, 1);

  int j, k;
  for (k = 2; k <= NUM_VALUES; k = 2 * k) {
    for (j = k >> 1; j > 0; j = j >> 1) {
      bitonic_sort_step<<<numBlocks, numThreads>>>(dev_values, j, k);
    }
```

```
  }
}
```

3. Bitonic Sort con varias GPUs

3.1. Implementación y explicación

En este caso a diferencia de la implementación con una sola GPU, tenemos la problemática de que si ordenamos cada segmento tal y como hemos hecho en el caso de una sola GPU, no aprovechamos el potencial de tener varias.
El motivo de que no se aproveche el potencial es que realmente, ordenaremos un trozo pero estos trozos no estarán ordenados entre ellos. Es decir, la concatenación de estos trozos casi nunca nos dará un conjunto de números bien ordenados y por lo tanto, terminaremos haciendo que una sola GPU haga la ordenación (lo cual es un desperdicio).

Para solucionar este problema, lo que haremos es que la memoria compartida entre las diversas GPU's que usemos, esté activada. El hecho de que se pueda compartir información entre GPU's juntamente con una adaptación correcta del algoritmo de Bitonic hasta ahora visto, nos dará lugar a un correcto algoritmo multigpu.

La adaptación que se hará en nuestro caso es de que hacemos las comparaciones entre posiciones de distintas GPUs para que así, progresivamente vayamos ordenando el vector (pensamos si el conjunto de valores de la GPU's estuvieran todas en una misma memoria).

El código pero tiene una problemática y es que la memoria que se puede compartir es limitada y por lo cual solo podemos llegar a utilizar este algoritmo para elementos de 4096 elementos (y una ampliación completa nos supondría comunicación para ir calculando poco a poco el resultado, pero no vemos cómo sería realmente).
Código

multiGPU.cu

```
__global__ void bitonicSortShared(int *dev_values)
{
    int tx = threadIdx.x;
    int bx = blockIdx.x;
    int index = blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    __shared__ int sh_values[SHARED_SIZE_LIMIT];
    sh_values[tx] = dev_values[index];
    sh_values[tx + (SHARED_SIZE_LIMIT/2)] = dev_values[index +
(SHARED_SIZE_LIMIT/2)];
    for (uint size = 2; size < SHARED_SIZE_LIMIT; size <<= 1) {
      uint ddd = (tx & (size / 2)) == 0;
```

```
    for (uint stride = size/2; stride > 0; stride >>= 1) {
        __syncthreads();
        uint pos = 2 * tx - (tx & (stride - 1));
        comparator(sh_values[pos], sh_values[pos + stride], ddd);
    }
  }
  uint ddd = ((bx&1) == 0); //    uint ddd = ((bx&1)==0);
  {
    for (uint stride = SHARED_SIZE_LIMIT/2; stride > 0; stride >>= 1) {
        __syncthreads();
        uint pos = 2 * tx - (tx & (stride - 1));
        comparator(sh_values[pos + 0], sh_values[pos + stride], ddd);
    }
  }
  __syncthreads();
  dev_values[index] = sh_values[tx];
  dev_values[index+(SHARED_SIZE_LIMIT/2)] =
sh_values[tx+(SHARED_SIZE_LIMIT/2)];
}
```

4. Comparación de las implementaciones

4.1. Comparación del rendimiento

No hemos podido medir los tiempos multigpu por culpa el servidor que se ha caído.

#Threads	1	2	4	8	16	32	64	128	256	512	1024
#Blocks	1	128	256	512	1024	2048	4096	8192	16384	32768	32768
#Elementos	1	256	1024	4096	16384	65536	262144	1048576	4194304	16777216	33554432
Tiempo CPU (s)	0	0.0041	0.093	0.337	0.8288	3.927	19.892	94.632	327.69		
Tiempo GPU (s)											0.511
Tiempo MultiGPU (4 GPU)											
SpeedUp GPU vs CPU											
SpeedUp MultiGPU vs GPU											

4.2. Conclusiones

Vistos los datos nuestras conclusiones son:

- El acceso a la memoria compartida es mucho más rápido que el acceso a memoria global pero tenemos el problema de que está limitada.
- Claramente los tiempos de GPU respecto a CPU son mucho mejores para empezar porque el código GPU está siendo paralelizado y los de CPU están siendo tirado en secuencial. Por otra parte, el potencial de GPU a nivel de cálculo es mayor
- Hacer la extensión para que un código en C pase a ser un código paralelo que se ejecuta en una GPU es muy fácil y sólo requiere de unas pocas horas de familiarización.

5. Bibliografía.

Webs/transparencias:

- http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm
- http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html
- https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158
- https://es.wikipedia.org/wiki/Ordenaci%C3%B3n_bit%C3%B3nica
- http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm
- http://www.cs.cmu.edu/~scandal/nesl/alg-sequence.html#bitonicsort
- http://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Mullapudi-Spring-2014-CSE633.pdf

- http://www.sci.brooklyn.cuny.edu/~amotz/BC-ALGORITHMS/PRESENTATIONS/networks-beamer.pdf
- https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=5&ved=0ahUKEwiU0b_vh_LMAhUPkRQKHdzyCAMQFghPMAQ&url=http%3A%2F%2Fwww.massey.ac.nz%2F~mjjohnso%2Fnotes%2F59735%2Fmyslides8.pdf&usg=AFQjCNFyxsvRy5YikOaWisUAGpsBnSNvEQ&cad=rja

Papers:

- https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjO-a7AjfLMAhVCxRQKHcA_D6QQFggkMAA&url=http%3A%2F%2Fdl.acm.org%2Fft_gateway.cfm%3Fid%3D2458524&usg=AFQjCNF2rDrYE0_BI1EL68EL8IMBIKoEwg&sig2=gwT7EAi7W07XMpdSw46efQ&cad=rja

- http://lap.epfl.ch/webdav/site/lap/shared/publications/YeApr10_HighPerformanceComparisonBasedSortingAlgorithmOnManyCoreGpus_IPDPS10.pdf