Sort

Trần Trung Kiên
ttkien@fit.hcmus.edu.vn

Cập nhật lần cuối: 15/12/2021

fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Nội dung

- Bài toán sort
- Radix Sort tuần tự
- Radix Sort song song

Bài toán sort

in	1	8	5	2	6	4	7	2

Stable sort	1	2	2	4	5	6	7	8

Unstable sort	1	2	2	4	5	6	7	8

Trước mắt, ta sẽ giới hạn việc sort với mảng số nguyên không dấu

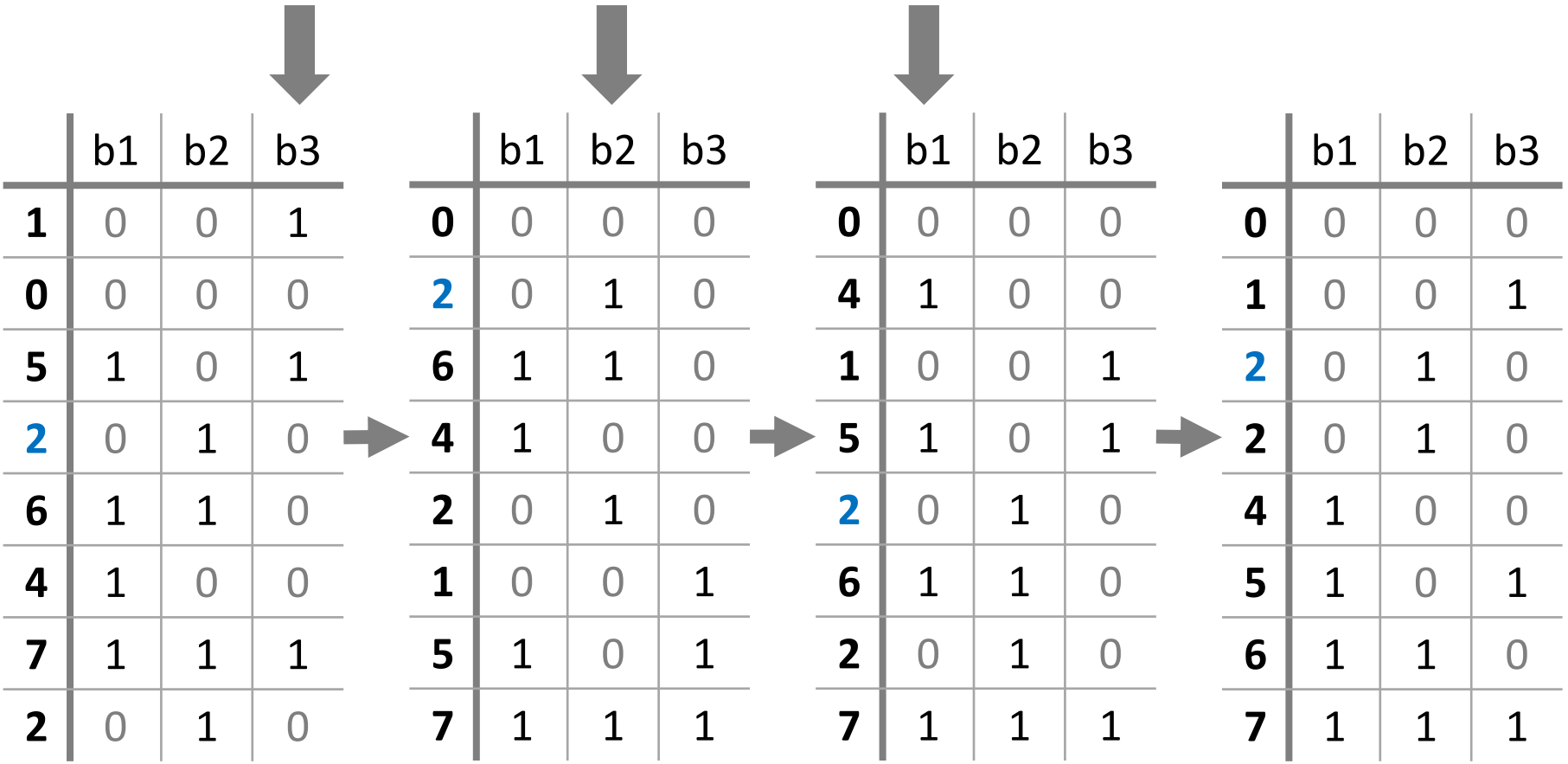
Nội dung

- Bài toán sort
- Radix Sort tuần tự
- Radix Sort song song

Radix Sort tuần tự

Duyệt từ bit b3 (bit ít quan trọng nhất) đến bit b1 (bit quan trọng nhất):

Sort các phần tử theo bit đang xét bằng một thuật toán sort mà stable



	b1	b2	b3
1	0	0	1
0	0	0	0
5	1	0	1
2	0	1	0
6	1	1	0
4	1	0	0
7	1	1	1
2	0	1	0

	b1	b2	b3
0	0	0	0
2	0	1	0
6	1	1	0
4	1	0	0
2	0	1	0
1	0	0	1
5	1	0	1
7	1	1	1

	b1	b2	b3
0	0	0	0
4	1	0	0
1	0	0	1
5	1	0	1
2	0	1	0
6	1	1	0
2	0	1	0
7	1	1	1

	b1	b2	b3
0	0	0	0
1	0	0	1
2	0	1	0
2	0	1	0
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

XONG!

Radix Sort tuần tự

- OK, Radix Sort chạy đúng
- Nhưng có hiệu quả không?

  Có, nếu ta có thể làm cho phần stable sort ở mỗi vòng lặp hiệu quả, ví dụ work = O(n)
  - Với số nguyên không dấu 32 bits,

    work của Radix Sort ≈ 32n = O(n)
  - Sẽ có thể hiệu quả hơn nữa nếu ta có thể xử lý k>1 bit ở mỗi vòng lặp (và vẫn giữ work ở mỗi vòng lặp là O(n))

    Để đơn giản, trong buổi học này, ta chỉ xét k=1 bit

Sort mảng nhị phân (ứng với $k = 1$ bit trong Radix Sort)

- Xét mảng nhị phân:

  `bits`: `0 1 1 0 1` (n elements)

  Làm sao để sort stable và hiệu quả?

- Ta sẽ dùng Counting Sort
  - Tính rank (chỉ số đúng trong mảng kết quả) của mỗi phần tử (work = O(n))

    `bits:` `0 1 1 0 1`

    `ranks:` `0 2 3 1 4`

    Rank của bits[i] =
    số lượng phần tử < bits[i]
    + số lượng phần tử = bits[i] và đứng trước bits[i]
  - Ghi mỗi phần tử xuống rank của nó trong mảng kết quả (work = O(n))

Sort mảng nhị phân (ứng với $k = 1$ bit trong Radix Sort)

- Xét mảng nhị phân:
  `bits`: 0 1 1 0 1 (n elements)
  Làm sao để sort stable và hiệu quả?
- Ta sẽ dùng Counting Sort
  - Tính rank (chỉ số đúng trong mảng kết quả) của mỗi phần tử (work = O(n))
    - Tính số lượng số 1 trước mỗi phần tử:

```
bits:          0 1 1 0 1
nOnesBefore:   0 0 1 2 2
```

Thực hiện exclusive scan

    - Tính rank:
      Nếu `bits[i]` là 0: `rank = i - nOnesBefore[i]`
      Nếu `bits[i]` là 1: `rank = nZeros + nOnesBefore[i]`
      Với `nZeros = n - nOnesBefore[n-1] - bits[n-1]`

```
bits:          0 1 1 0 1
ranks:         0 2 3 1 4
```

  - Ghi mỗi phần tử xuống rank của nó trong mảng kết quả (work = O(n))

Radix Sort tuần tự

Duyệt từ bit ít quan trọng nhất đến bit quan trọng nhất:

- Sort các phần tử theo bit đang xét bằng Counting Sort (stable và hiệu quả)

Live coding ….

Nội dung

- Bài toán sort
- Radix Sort tuần tự
- Radix Sort song song

Radix Sort tuần tự: song song hóa?

Song song hóa ✗

Duyệt từ bit ít quan trọng nhất đến bit quan trọng nhất:

- Sort các phần tử theo bit đang xét bằng Counting Sort (stable và hiệu quả)

Song song hóa ✓

Sort mảng nhị phân bằng Counting Sort: song song hóa?

- Xét mảng nhị phân:
  `bits`: 0 1 1 0 1 (n elements)
  Làm sao để sort stable và hiệu quả?
- Ta sẽ dùng Counting Sort
  - Tính rank (chỉ số đúng trong mảng kết quả) của mỗi phần tử (work = O(n))
    - Tính số lượng số 1 trước mỗi phần tử: ✓ Song song hóa

```
bits:          0 1 1 0 1
nOnesBefore:   0 0 1 2 2
```

Thực hiện exclusive scan

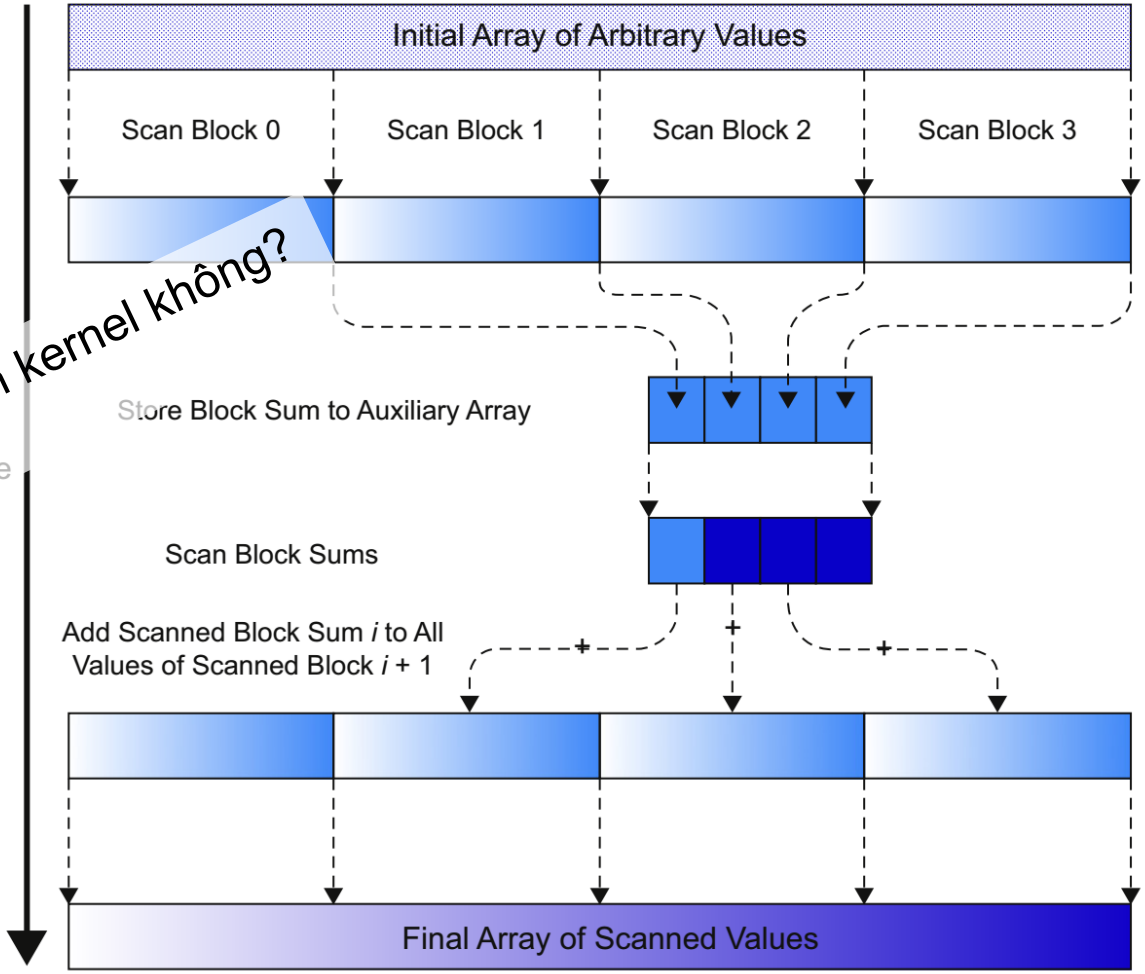
    - Tính rank: ✓ Song song hóa

Nếu `bits[i]` là 0: `rank = i - nOnesBefore[i]`
Nếu `bits[i]` là 1: `rank = nZeros + nOnesBefore[i]`
Với `nZeros = n - nOnesBefore[n-1] - bits[n-1]`

```
bits:          0 1 1 0 1
ranks:         0 2 3 1 4
```

  - Ghi mỗi phần tử xuống rank của nó trong mảng kết quả (work = O(n)) ✓ Song song hóa

Còn nhớ cách ta cài đặt song song tác vụ scan?



Ta có thể gom 3 bước này trong 1 hàm kernel không?
Ta có thể overlap 3 bước này không?

Nguồn ảnh: David B. Kirk et al. Programming Massively Parallel Processors

Scan toàn cục trong một hàm kernel

Block với chỉ số bi:

- Scan cục bộ
- Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1)

  Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)

  Block bi=0 thì chỉ bật tín hiệu thôi
- Hoàn thành phần việc còn lại: cộng tổng của bi block (0→bi-1) vào kết quả scan cục bộ của block mình

  Block bi=0 thì không cần làm bước này

Scan toàn cục trong một hàm kernel

Block với chỉ số bi:

- Scan cục bộ
- Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1)

  Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)

  Block bi=0 thì chỉ bật tín hiệu thôi
- Hoàn thành phần việc còn lại: cộng tổng của bi block (0→bi-1) vào kết quả scan cục bộ của block mình

  Block bi=0 thì không cần làm bước này

Một tình huống có thể xảy ra:
Block bi→bi+N được phân vào các chỗ trống trong các SM để thực thi,
và đang chờ kết quả của block bi-1
Block bi-1 đang chờ một chỗ trống trong các SM để có thể được thực thi
→ Deadlock ☹

Giải pháp: tính lại chỉ số bi của block, không dùng blockIdx.x

Scan toàn cục trong một hàm kernel

Block với chỉ số nào đó:

- Lấy chỉ-số-block bi mà có thứ tự
- Scan cục bộ
- Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1)

  Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)

  Block bi=0 thì chỉ bật tín hiệu thôi
- Hoàn thành phần việc còn lại: cộng tổng của bi block (0→bi-1) vào kết quả scan cục bộ của block mình

  Block bi=0 thì không cần làm bước này

Lấy chỉ-số-block bi có thứ tự

```
__device__ int bCount = 0;
…
__global__ scanKernel(…)
{
    …
    __shared__ int bi;
    if (threadIdx.x == 0)
        bi = atomicAdd(&bCount, 1);
    __syncthreads();
}
```

Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1). Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)
Block bi=0 thì chỉ bật tín hiệu thôi

```
__device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
        }
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1). Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)
Block bi=0 thì chỉ bật tín hiệu thôi

```
__device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
        }
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Compiler có thể optimize bằng cách cache các truy xuất bCount1 trong thanh ghi hoặc L1 cache

Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1). Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)
Block bi=0 thì chỉ bật tín hiệu thôi

```
__device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
        }
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Compiler có thể optimize bằng cách cache các truy xuất bSums trong thanh ghi hoặc L1 cache

Chờ cho tới khi thấy tín h -1 đã tính xong tổng của bi
block (0→bi-1). Lấy tổng lock mình, và bật tín hiệu để
block bi+1 biết block mình (0→bi)
Block bi=0 thì chỉ bật tín h

Đảm bảo compiler không cache các truy xuất bCount1 & bSums trong thanh ghi hay L1 cache

```
volatile __device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, volatile int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
        }
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Chờ cho tới khi thấy tín h -1 đã tính xong tổng của bi
block (0→bi-1). Lấy tổng lock mình, và bật tín hiệu để
block bi+1 biết block mình (0→bi)
Block bi=0 thì chỉ bật tín h

Đảm bảo compiler không cache các truy xuất bCount1 & bSums trong thanh ghi hay L1 cache

```
volatile __device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, volatile int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
        }   Có chắc bSums sẽ được cập nhật trước bCount1?
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Chờ cho tới khi thấy tín h -1 đã tính xong tổng của bi
block (0→bi-1). Lấy tổng lock mình, và bật tín hiệu để
block bi+1 biết block mình (0→bi)
Block bi=0 thì chỉ bật tín h

Đảm bảo compiler không cache các truy xuất bCount1 & bSums trong thanh ghi hay L1 cache

```
volatile __device__ int bCount1 = 0;
…
__global__ scanKernel(int * in, int n, int * out, volatile int * bSums)
{
    …
    // bSums chứa tổng cục bộ của mỗi block,
    // block bi đã tính xong bSums[bi]
    if (threadIdx.x == 0)
    {
        if (bi > 0)
        {
            while (bCount1 < bi) {} // Chờ block bi-1
            bSums[bi] += bSums[bi-1]; // Tính tổng của bi+1 block (0→bi)
            __threadfence(); // Đảm bảo bSums được cập nhật trước bCount1
        }
        bCount1 += 1; // Bật tín hiệu để block bi+1 biết
    }
    __syncthreads();
    …
}
```

Đọc thêm:
- Document về __threadfence
- Document về volatile

Scan toàn cục trong một hàm kernel

Block với chỉ số nào đó:

- Lấy chỉ-số-block bi mà có thứ tự
- Scan cục bộ
- Chờ cho tới khi thấy tín hiệu của block bi-1 cho biết block bi-1 đã tính xong tổng của bi block (0→bi-1)

  Lấy tổng này, cộng tổng này vào tổng của block mình, và bật tín hiệu để block bi+1 biết block mình đã tính xong tổng của bi+1 block (0→bi)

  Block bi=0 thì chỉ bật tín hiệu thôi
- Hoàn thành phần việc còn lại: cộng tổng của bi block (0→bi-1) vào kết quả scan cục bộ của block mình

  Block bi=0 thì không cần làm bước này

Tuần tự giữa các block

Tuần tự giữa các block

Inclusive scan $\overset{?}{\rightarrow}$ exclusive scan

Cài đặt Radix Sort song song, dùng scan toàn cục trong một hàm kernel

HW4 ;-)

Radix Sort cho số nguyên có dấu

- Bit dấu là bit quan trọng nhất
  - Bit dấu = 0: số dương

    Số nguyên có dấu = số nguyên không dấu
  - Bit dấu = 1: số âm

    Số nguyên có dấu = số nguyên không dấu

    $- 2^{\text{số-bit-biểu-diễn}}$
- Nếu ta dùng Radix Sort cho số nguyên không dấu để sort số nguyên có dấu thì sẽ sai
- Một giải pháp:
  - Chuyển số nguyên có dấu thành số nguyên không dấu
  - Chạy Radix Sort cho số nguyên không dấu
  - Chuyển kết quả ngược lại số nguyên có dấu

Radix Sort cho số thực

- Cần hiểu về cách số thực được biểu diễn dưới dạng bit
- Ý tưởng tương tự số nguyên có dấu:
  - Chuyển số thực thành số nguyên không dấu
  - Chạy Radix Sort cho số nguyên không dấu
  - Chuyển kết quả ngược lại số thực