Parallel Programming

Prefix sum (scan)

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

Overview

- The “sort” task
- Sequential Radix Sort
- Parallel Radix Sort

The “sort” task

in	1	8	5	2	6	4	7	2
<u>Stable</u> sort	1	2	2	4	5	6	7	8
<u>Unstable</u> sort	1	2	2	4	5	6	7	8

We will focus on input array of unsigned ints

Overview

- The “sort” task
- Sequential Radix Sort
- Parallel Radix Sort

Sequential Radix Sort

Loop from bit b3 (least significant bit) to b1 (most significant bit):

Sort elements w.r.t. the current bit using a stable sort

	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

DONE!

Sequential Radix Sort

- OK, Radix Sort works
- But is it efficient?

  Yes, if we can make the stable sort in each loop efficient, e.g. work = $O(n)$

  - With unsigned int (32 bits),

    Radix Sort's work ≈ $32n = O(n)$
  - It's potentially even more efficient if we process $k>1$ bits in each loop (and still keep the work in each loop at $O(n)$)

    For simplicity, in this lecture, we just consider $k=1$ bit

Sort a binary array (corresponding to $k = 1$ bit in Radix Sort)

- Consider a binary input array:
  `binIn`: 0 1 1 0 1 (n elements)
  How to sort stably and efficiently?
- We will use Counting Sort
  - Compute the rank (the correct index in the output array) of each element (work = O(n))

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

    Rank of `binIn[i]` =
    \# elements < `binIn[i]`
    \+ # elements before `binIn[i]` and = `binIn[i]`

  - Write each element to its rank in the output array (work = O(n))

Sort a binary array (corresponding to $k = 1$ bit in Radix Sort)

- Consider a binary input array:
  `binIn`: `0 1 1 0 1` (`n` elements)
  How to sort stably and efficiently?
- We will use Counting Sort
  - Compute the rank (the correct index in the output array) of each element (work = O(n))
    - Compute # ones before each element:
      `binIn`: `0 1 1 0 1`
      `nOnesBefore`: `0 0 1 2 2`
      Do exclusive scan
    - Compute rank:
      if `binIn[i]` is 0: `rank = i - nOnesBefore[i]`
      if `binIn[i]` is 1: `rank = nZeros + nOnesBefore[i]`
      With `nZeros = n - nOnesBefore[n-1] - binIn[n-1]`
      `binIn`: `0 1 1 0 1`
      `ranks`: `0 2 3 1 4`
  - Write each element to its rank in the output array (work = O(n))

Sequential Radix Sort

Loop from Least Significant Bit to Most Significant Bit:

- Sort elements w.r.t the current bit using Counting Sort (stably and efficiently)

Let’s implement this …

Overview

- The “sort” task
- Sequential Radix Sort
- Parallel Radix Sort

Sequential Radix Sort: parallelize?

Loop from Least Significant Bit to Most Significant Bit:

Sort elements w.r.t the current bit using Counting Sort

Parallelize ✗

Parallelize ✓

Sort a binary array using Counting Sort: parallelize?

- Consider a binary input array:
  `binIn`: 0 1 1 0 1 (n elements)
  How to sort stably and efficiently?
- We will use Counting Sort
  - Compute the rank (the correct index in the output array) of each element (work = O(n))
    - Compute # ones before each element:
      `binIn:` 0 1 1 0 1
      `nOnesBefore:` 0 0 1 2 2
      Do exclusive scan
    - Compute rank:
      if `binIn[i]` is 0: `rank = i - nOnesBefore[i]`
      if `binIn[i]` is 1: `rank = nZeros + nOnesBefore[i]`
      With `nZeros = n - nOnesBefore[n-1] - binIn[n-1]`
      `binIn:` 0 1 1 0 1
      `ranks:` 0 2 3 1 4
  - Write each element to its rank in the output array (work = O(n))

Parallelize✓

Parallelize✓

Parallelize✓

Remember how do we implement scan in parallel?

Can we do all 3 phases in a single kernel?
Can we overlap these 3 phases?

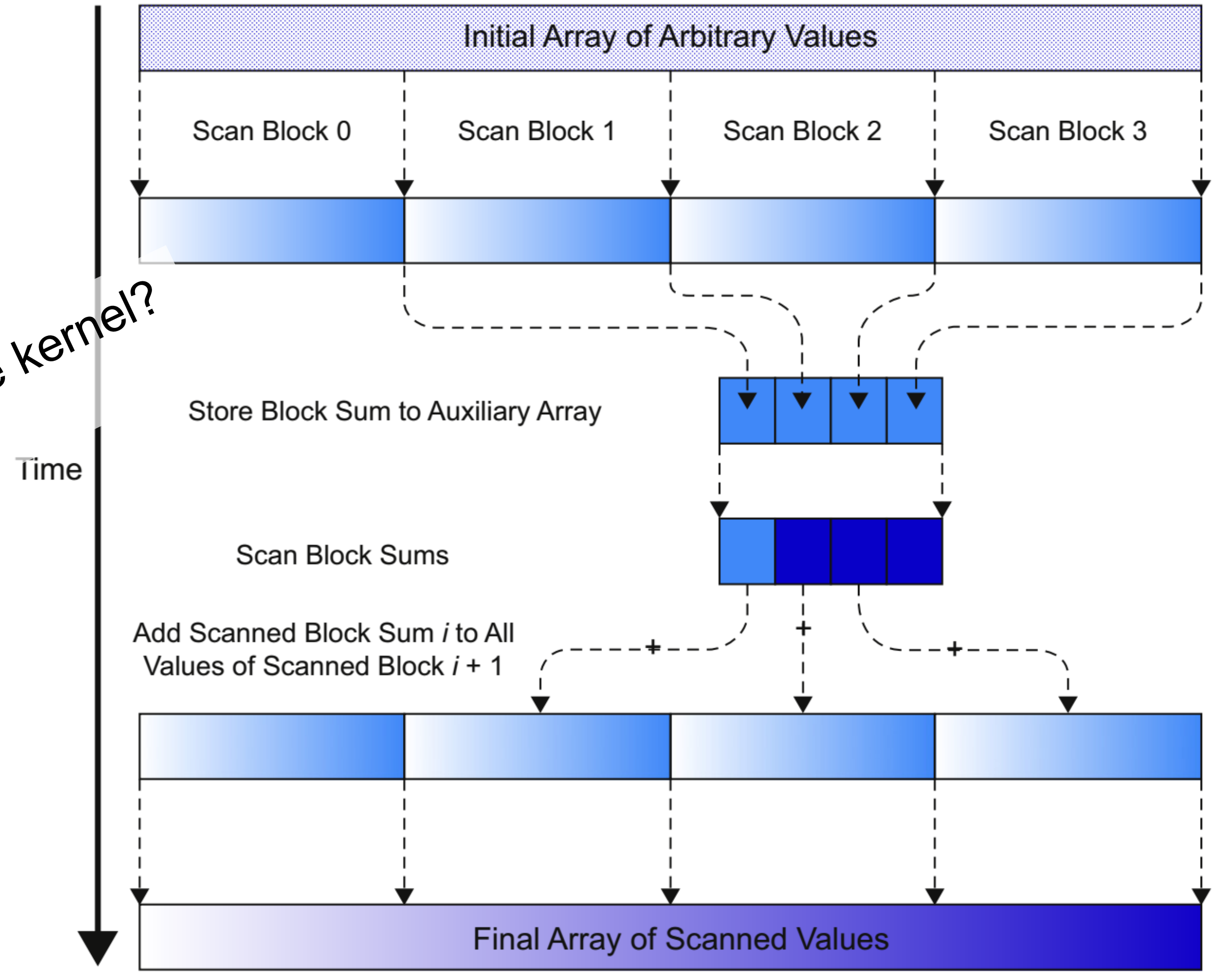

Image source: David B. Kirk et al. Programming Massively Parallel Processors

Global scan in a single kernel

Block with index bi:

- Scan locally
- Wait until seeing the sign indicating block bi-1 has computed the sum of bi blocks (0→bi-1)

  Get this sum, add this sum to block bi's local sum, and turn on the sign indicating that block bi has computed the sum of bi+1 blocks (0→bi)

  (Block bi=0 only needs to turn on the sign)
- Finish the rest of work: add the sum of bi blocks (0→bi-1) to block bi's local scan

  (Block bi=0 will not do this step)

Global scan in a single kernel

Block with index bi:

- Scan locally
- Wait until seeing the sign indicating block bi-1 has computed the sum of bi blocks (0→bi-1)
- Get this sum, add this sum to block bi's local sum, and turn on the sign indicating that block bi has computed the sum of bi+1 blocks (0→bi)

  (Block bi=0 only needs to turn on the sign)
- Finish the rest of work: add the sum of bi blocks (0→bi-1) to block bi's local scan

  (Block bi=0 will not do this step)

A possible situation:
Blocks bi→bi+N are assigned to available slots in SM, and wait for the result from block bi-1
Block bi-1 waits for an available slot in SM
→ Deadlock ☹

Solution: recompute block index bi, don't tie it with blockIdx.x

Global scan in a single kernel

Block with index bi:

- Get in-order block index bi
- Scan locally
- Wait until seeing the sign indicating block bi-1 has computed the sum of bi blocks (0→bi-1)

  Get this sum, add this sum to block bi's local sum, and turn on the sign indicating that block bi has computed the sum of bi+1 blocks (0→bi)

  (Block bi=0 only needs to turn on the sign)
- Finish the rest of work: add the sum of bi blocks (0→bi-1) to block bi's local scan

  (Block bi=0 will not do this step)

Get in-order block index bi

- *blkCount1*: dùng để gán block ID (bid) mới. Giá trị đầu = 0
    - Block đầu tiên chạy sẽ có giá trị (bid = 0), *blkCount1*++
    - Block thứ 2 sẽ có bid = 1, *blkCount1*++
    - …
- Chỉ thread 0 cần tính phần này vào biến share. Sau đó các thread khác lấy giá trị share này vào biến register của mình.

```
__device__ int blkCount1 = 0;
__device__ int blkCount2 = 0;
//…
if (threadIdx.x == 0)
        s_data[0] = atomicAdd(&blkCount1,1);
 __syncthreads();
int bid = s_data[0];
```

Get in-order block index bi

```
__device__ int blkCount1 = 0;
__device__ int blkCount2 = 0;
//...
if (threadIdx.x == 0){
       blkSums[bid] = s_data[2 * blockDim.x - 1];
       if (bid > 0){
              while (atomicAdd(&blkCount2, 0) < bid) {}
              s_data[blockDim.x * 2] = blkSums[bid - 1];
              blkSums[bid] += s_data[blockDim.x * 2];
              __threadfence();
       }
       atomicAdd(&blkCount2, 1);
}
__syncthreads();
```

Read more:
- Document about __threadfence
- Document about volatile

Global scan in a single kernel

Block with index bi:

Serialize between blocks

Serialize between blocks

- Get in-order block index bi
- Scan locally
- Wait until seeing the sign indicating block bi-1 has computed the sum of bi blocks (0→bi-1)

  Get this sum, add this sum to block bi's local sum, and turn on the sign indicating that block bi has computed the sum of bi+1 blocks (0→bi)

  (Block bi=0 only needs to turn on the sign)
- Finish the rest of work: add the sum of bi blocks (0→bi-1) to block bi's local scan

  (Block bi=0 will not do this step)

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

Implement parallel Radix Sort using global scan in a single kernel

The upcoming HW4 ;-)

Radix Sort for signed ints

- Sign bit is MSB (Most Significant Bit)
  - MSB = 0: positive number

    Signed int = unsigned int
  - MSB = 1: negative number

    Signed int = unsigned int - $2^{\#bits\text{-}of\text{-}signed\text{-}int}$
- If we use Radix Sort for unsigned ints, it'll be wrong
- One solution:
  - Convert signed ints to unsigned ints
  - Run Radix Sort for unsigned ints
  - Convert results back to signed ints

Radix Sort for floats

- Need to understand how floats are represented
- Idea is similar to signed ints:
  - Convert floats to unsigned ints
  - Run Radix Sort for unsigned ints
  - Convert results back to floats

Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022
- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- [3] Illinois GPU course

https://wiki.illinois.edu/wiki/display/ECE408/ECE408+Home

THE END