

CPU Radix sort

For least significant digit to most significant digit, sorting number one by one

- Initialize 0-9 to has a corresponding queue
- scan array from left to right; For each element, extract the digit
- enqueue each element to the corresponding queue by the digit
- For 0-9, dequeue elements in queues to form a new array

e.g. use radix sort to sort 21,11,28,15

- Extract 1st round digit: 21,11,28,15
- 1's queue: 21,11; 5's queue: 15; 8's queue:28;
- 1st round result: 21,11,25,28
- Extract 2nd round digit: 21,11,28,15
- 1's queue: 11,15; 2's queue:21,28;
- 2nd round result: 11,15,21,28

The black part can be done in parallel

The red part can only be carried out one by one

We can use the exclusive prefix sum algorithm to get rid of using queue and thus make the algorithm more suitable for GPU. The key idea is to compute each element's offset in the queue directly.

Exclusive Prefix Sum

Definition: The sum of elements before it(not include itself)

- E.g. input 1,2,3,4 -> 0,1,2,3

Use exclusive prefix sum in directly computing the offset in each round's output

- E.g 1st round digit: 21,11,28,15; how 15,11 know their offset in output?
- 15's offset=1's queue length(2) + 15's position in 5's queue(0) = 2
- 11's offset=0 + 11's position in 1's queue(1) = 1
- Create a mask for digit 1, 1 for end with 1, 0 otherwise: 1, 1 ,0 ,0
- Exclusive prefix sum of 1, 1 ,0 ,0 = 0,1,2,2
- 1's queue length = last element in exclusive prefix sum(2) + 1

We have two parts: one is the sum of queue length; one is the position in the queue

- ☐ If we create 0/1 mask for each digit, position in the queue can be solved as an exclusive prefix sum problem
- ☐ We can compute the queue length by adding 1 to the last element in this queue's exclusive prefix sum. Then the sum of queue length is equal to the exclusive prefix sum over the queue length

GPU Radix sort

For least significant digit to most significant digit

- For each element, extract the digit and put 0/1 in the corresponding mask
- Use exclusive prefix sum to compute the offset in each round's output directly
- Put each element in the output array according to the offset

Blelloch algorithm for Exclusive Prefix Sum

Idea: divide and conquer

Subproblem: add two adjacent number, forming a new array which is half the size

If we can solve the prefix sum of subproblem, we can solve the original problem by replacing and adding

Recursive termination: when input is one element array return [0]

```
In [43]: import random
import math
import copy

In [44]: data = [random.randint(0,16) for i in range(16)]
data
Out[44]: [12, 4, 12, 16, 16, 2, 6, 3, 7, 7, 14, 16, 14, 12, 16, 3]

In [51]: def prefix_sum(data,n):
    print('in', data, n)
    if n == 1:
        print('out', [0], n)
        return [0]
    new_array = [data[2*i] + data[2*i+1] for i in range(n//2)]
    sub_prefix_sum = prefix_sum(new_array,n//2)
    for i in range(n//2):
        temp = data[2*i]
        data[2*i] = sub_prefix_sum[i]
        data[2*i+1] = data[2*i] + temp
    print('out', data, n)
    return data

In [52]: prefix_sum(copy.deepcopy(data),16)

in [12, 4, 12, 16, 16, 2, 6, 3, 7, 7, 14, 16, 14, 12, 16, 3] 16
in [16, 28, 18, 9, 14, 30, 26, 19] 8
in [44, 27, 44, 45] 4
in [71, 89] 2
in [160] 1
out [0] 1
out [0, 71] 2
out [0, 44, 71, 115] 4
out [0, 16, 44, 62, 71, 85, 115, 141] 8
out [0, 12, 16, 28, 44, 60, 62, 68, 71, 78, 85, 99, 115, 129, 141, 157] 16
Out[52]: [0, 12, 16, 28, 44, 60, 62, 68, 71, 78, 85, 99, 115, 129, 141, 157]
```

- e.g. Input: 1,2,3,4,5,6,7,8
- Expect: 0,1,3,6,10,15,21,28
- Adding: 3,7,11,15
- Prefix sum: 0, 3, 10, 21
- Replace even: 0, 2, 3, 4, 10, 6, 21, 8
- Add odd: 1, 3, 5, 7
- Result: 0, 1, 3, 6, 10, 15, 21, 28

Loop transformation

We need to transfer recursive calling to loop so that it can work on GPU

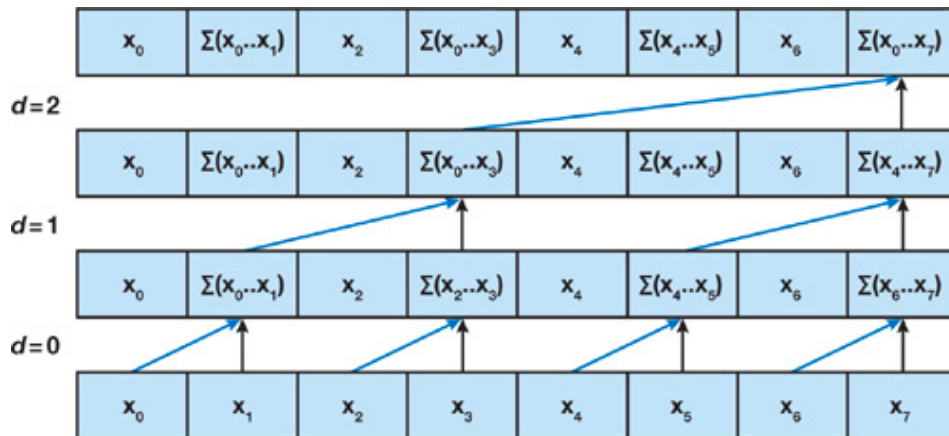
```
1. def prefix_sum(data,n):
2.     print('in', data, n)
3.     if n == 1:
4.         print('out', [0], n)
5.         return [0]
6.     new_array = [data[2*i] + data[2*i+1] for i in range(n//2)]
7.     sub_prefix_sum = prefix_sum(new_array,n//2)
8.     for i in range(n//2):
9.         temp = data[2*i]
10.        data[2*i] = sub_prefix_sum[i]
11.        data[2*i+1] = data[2*i] + temp
12.    print('out', data, n)
13.    return data
```

```

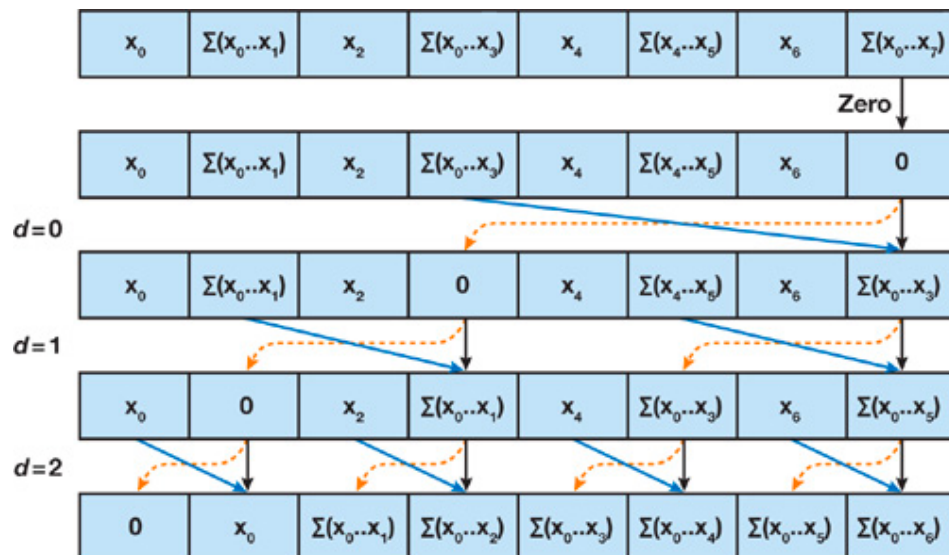
1. int offset = 0;
2. for (int d = numElems >> 1; d > 0; d >>= 1)
3. {
4.     __syncthreads();
5.
6.     if (thid < d)
7.     {
8.         int ai = (dthid1 << offset) - 1;
9.         int bi = (dthid2 << offset) - 1;
10.        s_out[CONFLICT_FREE_OFFSET(bi)] += s_out[CONFLICT_FREE_OFFSET(ai)];
11.    }
12.    offset++;
13. }
14.
15. if (thid == 0)
16. {
17.     d_block_sums[blockIdx.x] = s_out[CONFLICT_FREE_OFFSET(numElems - 1)];
18.     s_out[CONFLICT_FREE_OFFSET(numElems - 1)] = 0;
19. }
20.
21. for (int d = 1; d < numElems; d <= 1)
22. {
23.     offset--;
24.     __syncthreads();
25.
26.     if (thid < d)
27.     {
28.         int ai = (dthid1 << offset) - 1;
29.         int bi = (dthid2 << offset) - 1;
30.         uint32_t temp = s_out[CONFLICT_FREE_OFFSET(ai)];
31.         s_out[CONFLICT_FREE_OFFSET(ai)] = s_out[CONFLICT_FREE_OFFSET(bi)];
32.         s_out[CONFLICT_FREE_OFFSET(bi)] += temp;
33.     }
34. }
35. __syncthreads();

```

Firstly, we unroll every operation before the recursive call, corresponding to python code line 2 to line 11 and CUDA code line 15 to line 34



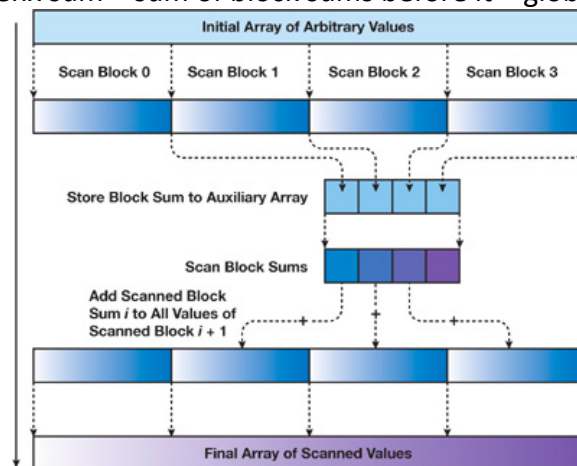
Secondly, we unroll every operation after the recursive call, corresponding to python code line 8 to line 7 and CUDA code line 2 to line 13



Inter-block Prefix Sum

Blelloch algorithm requires __syncthread. However, at grids level, we do not have synchronization. We solve this issue by adding local prefix sum with the sum of block sums before it so that we can get the global prefix sum.

local prefix sum + sum of block sums before it = global prefix sum



To compute the sum of block sums before it, we firstly store each block's total sum in an auxiliary array. We perform an exclusive prefix scan on it and get each blocks' prefix sum. Then we add each intra-block prefix sum with the prefix sum of this block.

Avoid bank conflict

Blelloch algorithm has stride memory access

- 1st 0, 2...16...30 thread 8 bank conflict with thread 0 => 2-degree bank conflict
- 2nd 1, 5...17...33...49...61=> 4-degree bank conflict
- 3rd 1, 9...17...33...49...65...81...97...113...121=> 8-degree bank conflict

Padding 1 element every 16 elements, so that we do not have share memory bank conflict.

- 1st 0, 2...17...31
- Bank 0,2,4,...,1,3,15
- 2nd 1, 5...18...35...52...64
- Bank 1,5,...,2...3...4...0
- 3rd 1, 9...18...35...52...69...86...103...120...128

- Bank 1,9,...2...3...4...5...6...7...8...0

global memory coalescing

- Input:32,23,56,37,89,41.....
- Digit: 2,3,6,7,9,1
- Assume 0-9 counter are all 100
- Offset:200,300,600,700,900,100

We have very sparse global memory access if we use global offset to sort the array

- Sort in-block data using in-block offset, write back to global memory
- Use prefix sum to get the final offset, move data to the final position
- When do the final move:
 - 10,20,30,40.....maps to continuous global memory

e.g.

We make a 1024 thread block sort a 2048 element. We sort 2 bits at a time. In theory, we should have, on average, 512 continuous global memory writing.

Implementation

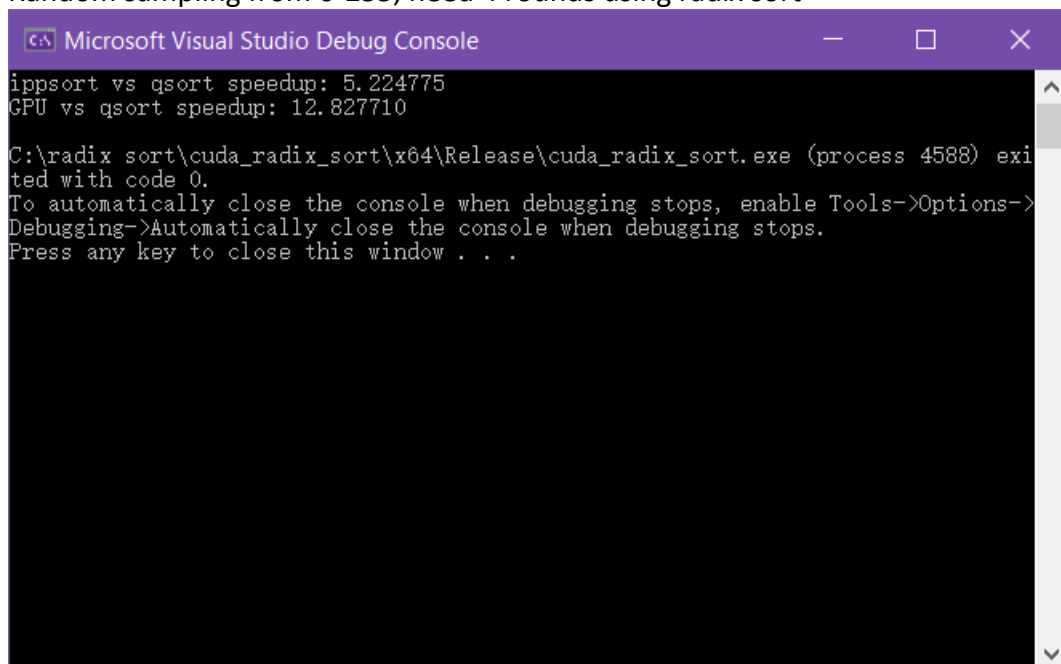
The only thing we need to notice is that we need to use registers to store elements when we shift elements with regard to the prefix sum. Since only one warp is running at a time, the data one thread need can be altered by other warps running before it reads. We need to load the data into registers, call `__syncthreads`, and then write registers back to shared memory.

The whole visual studio project is in the 7z file. You need to install IPP and CUDA to compile it.

Result

I add radix sort in the Intel Integrated Performance Primitives(IPP) and qsort in stdlib.h as baseline. I sort on 32M unsigned int. I log the speedup of IPP radix sort using CPU against qsort and my GPU radix sort against qsort. I change the random number value span to create multiple sets of experiments. Obviously, the small the span, the bigger the advantage radix sort have over quick sort. The quick sort is $O(n\log n)$ while the radix sort is $O(n)$

- Random sampling from 0-255, need 4 rounds using radix sort

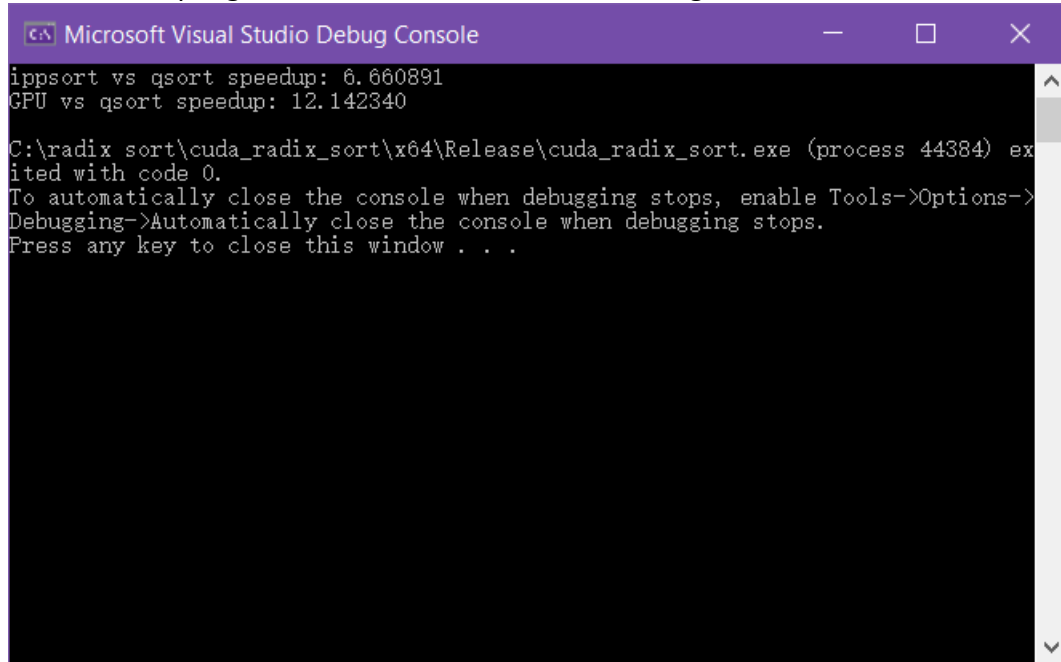


```

Microsoft Visual Studio Debug Console
ippsort vs qsort speedup: 5.224775
GPU vs qsort speedup: 12.827710

C:\radix sort\cuda_radix_sort\x64\Release\cuda_radix_sort.exe (process 4588) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
  
```

- Random sampling from 0-65536, need 8 rounds using radix sort



```

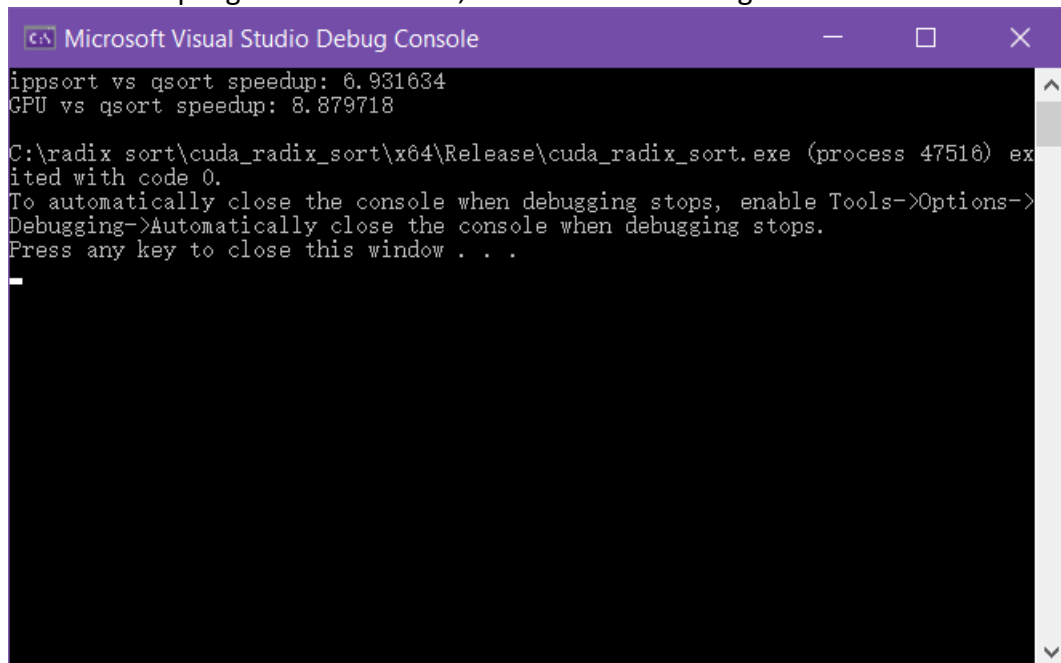
Microsoft Visual Studio Debug Console

ippsort vs qsort speedup: 6.660891
GPU vs qsort speedup: 12.142340

C:\radix sort\cuda_radix_sort\x64\Release\cuda_radix_sort.exe (process 44384) ex
ited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->
Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

- Random sampling from 0-1677216, need 12 rounds using radix sort



```

Microsoft Visual Studio Debug Console

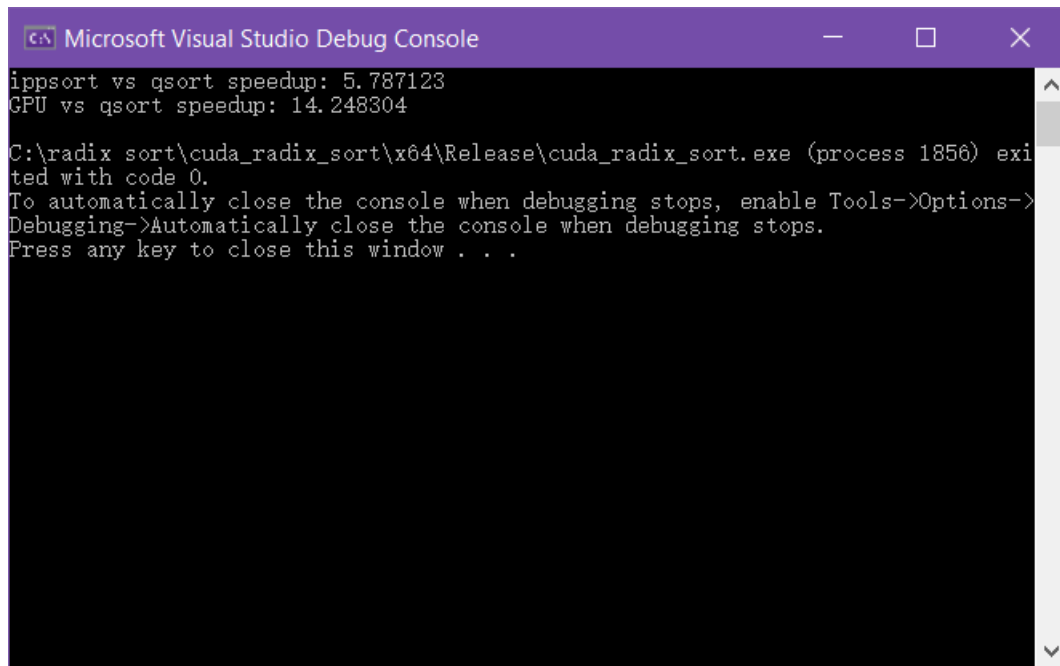
ippsort vs qsort speedup: 6.931634
GPU vs qsort speedup: 8.879718

C:\radix sort\cuda_radix_sort\x64\Release\cuda_radix_sort.exe (process 47516) ex
ited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->
Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

I also try to determine whether my two optimizations are truly useful.

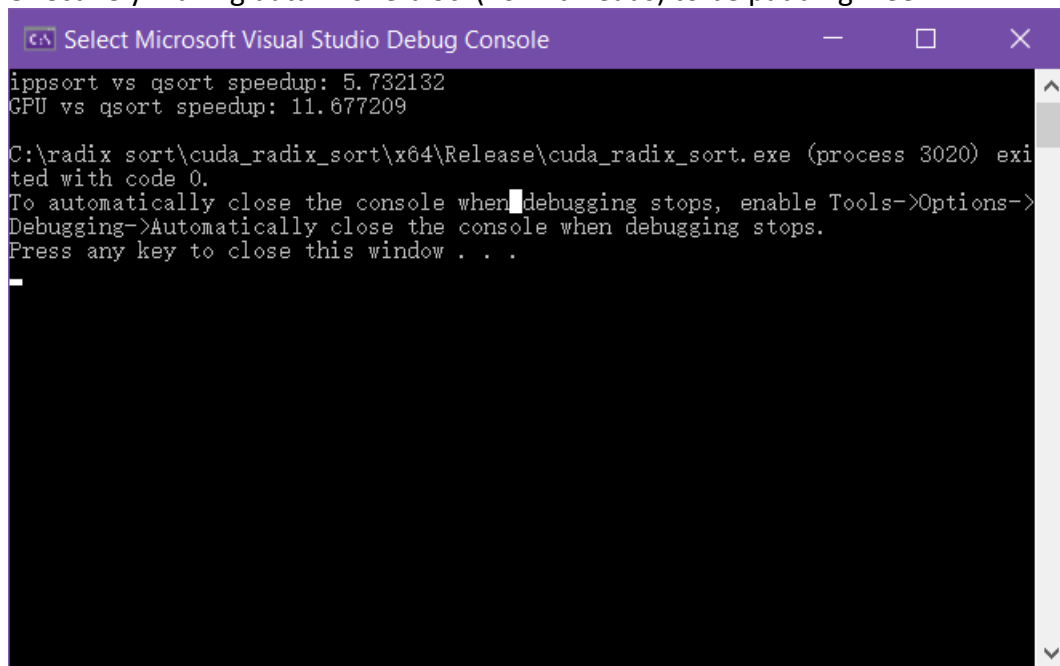
Without global memory coalescing, my GPU radix sort actually speedup when random sampling from 0-255. I think this is caused by only sorting 2 bits at a time. With 32 threads in a warp, we will still have 8 continuous global memory access on average thanks to hardware global memory coalescing. Maybe this technique will work when we sort more bits at a time. But this will require one block to take more hardware resources and may hurt occupancy.



```
Microsoft Visual Studio Debug Console
ippsort vs qsort speedup: 5.787123
GPU vs qsort speedup: 14.248304

C:\radix sort\cuda_radix_sort\x64\Release\cuda_radix_sort.exe (process 1856) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Without padding to avoid shared memory bank conflicts, my GPU radix sort slow down a little. We disable padding by set padding frequency to be one element every 4096 element, effectively making data in one block(1024 threads) to be padding-free.



```
Select Microsoft Visual Studio Debug Console
ippsort vs qsort speedup: 5.732132
GPU vs qsort speedup: 11.677209

C:\radix sort\cuda_radix_sort\x64\Release\cuda_radix_sort.exe (process 3020) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Reference

Most of my idea comes from

- https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
 - How to avoid bank conflict
- <https://github.com/mark-poscablo/gpu-radix-sort>
 - How to coalescing global memory access
 - Use as correctness checking. I replace one function at a time, if the result is wrong, then I know where the problem is