

# BÁO CÁO ĐỒ ÁN CUỐI KỲ

Môn Lập trình song song trên GPU

Chủ đề: **RADIX SORT**

Giảng viên hướng dẫn: Trần Trung Kiên

Sinh viên thực hiện

Trần Thái Quang Hoàng 1412189

Võ Phương Hòa 1412192

\*\*\*\*\*

## I. Môi trường cài đặt

Hệ điều hành: Ubuntu 16.04

GPU: Nvidia Geforce 820M (cc: 2.1)

## II. Quá trình cài đặt

### 1. Phiên bản tuần tự

Ý tưởng cài đặt: Thực hiện radix sort trên biểu diễn nhị phân của số nguyên không âm kết hợp các kỹ thuật histogram, scan và scatter.

Xét trên dữ liệu nguyên không âm  $2^n$  bit, radix sort trên từng digit (bit) (theo thứ tự từ least significant bit (LSB) tới most significant bit (MSB)).

Mỗi vòng lặp thực hiện trên k-bit ( $k = 2^i, 0 \leq i \leq n - 1$ ).

Trong mỗi vòng lặp:

**B1**: Histogram để tìm số lần xuất hiện của mỗi digit.

**B2**: Exclusive scan trên histogram.

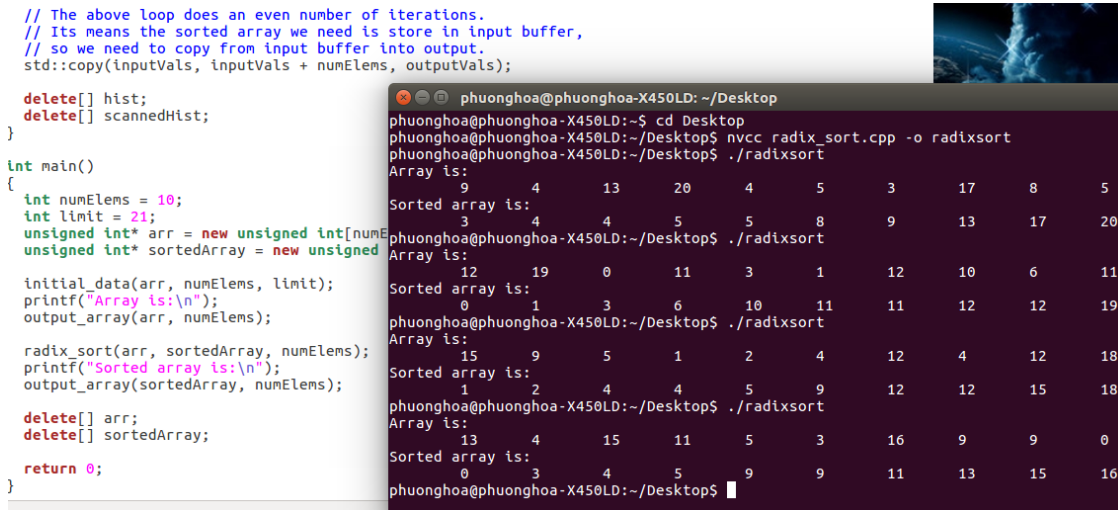
**B3**: Tìm rank của digit

VD: [0 1 0 0 1 0 0 1]

rank -> [0 0 1 2 1 3 4 2]

**B4**: Lấy kết quả từ B2 và B3 để thực hiện scatter dữ liệu vào vị trí đúng

File cài đặt radix sort: **radix\_sort.cpp**  
 Kết quả thực thi :



```
// The above loop does an even number of iterations.
// Its means the sorted array we need is store in input buffer,
// so we need to copy from input buffer into output.
std::copy(inputVals, inputVals + numElems, outputVals);

delete[] hist;
delete[] scannedHist;
}

int main()
{
    int numElems = 10;
    int limit = 21;
    unsigned int* arr = new unsigned int[numElems];
    unsigned int* sortedArray = new unsigned int[numElems];

    initial_data(arr, numElems, limit);
    printf("Array is:\n");
    output_array(arr, numElems);

    radix_sort(arr, sortedArray, numElems);
    printf("Sorted array is:\n");
    output_array(sortedArray, numElems);

    delete[] arr;
    delete[] sortedArray;

    return 0;
}
```

```
phuonghoa@phuonghoa-X450LD: ~/Desktop
phuonghoa@phuonghoa-X450LD:~/Desktop$ cd Desktop
phuonghoa@phuonghoa-X450LD:~/Desktop$ nvcc radix_sort.cpp -o radixsort
phuonghoa@phuonghoa-X450LD:~/Desktop$ ./radixsort
Array is:
9 4 13 20 4 5 3 17 8 5
Sorted array is:
3 4 4 5 5 8 9 13 17 20
phuonghoa@phuonghoa-X450LD:~/Desktop$ ./radixsort
Array is:
12 19 0 11 3 1 12 10 6 11
Sorted array is:
0 1 3 6 10 11 11 12 12 19
phuonghoa@phuonghoa-X450LD:~/Desktop$ ./radixsort
Array is:
15 9 5 1 2 4 12 4 12 18
Sorted array is:
1 2 4 4 5 9 12 12 15 18
phuonghoa@phuonghoa-X450LD:~/Desktop$ ./radixsort
Array is:
13 4 15 11 5 3 16 9 9 0
Sorted array is:
0 3 4 5 9 9 11 13 15 16
phuonghoa@phuonghoa-X450LD:~/Desktop$
```

File cài đặt radix sort (tương tự file reference\_calc.cpp trong problem set 4) dùng để so sánh thời gian thực thi với phép sort trong thrust: **student\_reference\_calc.cpp**

Bảng so sánh thời gian chạy radix sort tuần tự và song song (dùng thuật toán sort trong thư viện thrust) (đơn vị: ms (milliseconds))

		histogram's time	scan's time	scatter's time	total time
tuần tự	k-bit = 1	12.1913	0.0010	17.0665	29.5771
	k-bit = 4	2.6493	0.0010	4.1289	7.1289
	k-bit = 8	1.3089	0.0012	3.7961	5.3708
sort (in thrust)					3.3823

\*\*\* Total time luôn lớn hơn tổng thời gian histogram, scan và scatter vì total time là thời gian thực thi toàn bộ file tuần tự (bao gồm cả thời gian thực thi các câu lệnh khác trong file).

## 2. Phiên bản song song

### 2.1. Phiên bản 1

Ý tưởng cài đặt: lấy ý tưởng từ phiên bản tuần tự, thực hiện radix sort trên từng digit. Tức là, phiên bản này chỉ xét trong trường hợp đơn giản nhất: **nBits = 1**.

### Thiết kế:

Xây dựng từng kernel cho mỗi bước, mỗi kernel có một blockSize và gridSize riêng.

- **B1: Histogram**

Sử dụng 2 kernel: 1 kernel để tính digit (takecurrentDigit kernel) và 1 kernel thực hiện histogram trên digit vừa tìm được.

VD: xét dữ liệu  $A = [1\ 3\ 5\ 2\ 6\ 4]$  trong vòng lặp thứ nhất

digit là  $[1\ 1\ 1\ 0\ 0\ 0]$ , tiến hành histogram trên digit này, ta được  $[3\ 3]$ .

\*\*\*Không exclusive scan trên histogram vì số bin = 2 (rất nhỏ và không cần thiết phải scan). Với số bin = 2, khai báo histogram có 3 phần tử, khởi tạo bằng 0, truyền tham số là (histogram + 1) vào kernel histogram, ta sẽ có luôn kết quả exclusive scan trên histogram là  $[0\ 3\ 3]$  (bỏ phần tử cuối vì không cần thiết).

- **B2 : Exclusive scan trên digit (để chuẩn bị cho bước tìm rank).**

Kernel scan là inclusive. Để chuyển sang exclusive scan, khai báo một con trỏ exScan trỏ tới vùng nhớ có kích thước  $m + 1$  ( $m$  là kích thước của  $A$ ), khởi tạo bằng 0 cho toàn bộ vùng nhớ. Lúc gọi kernel scan, gọi con trỏ ( $\text{exScan} + 1$ ).

VD : xét tiếp ví dụ trên, exScan giữ kết quả là  $[0\ 1\ 2\ 3\ 3\ 3]$  (ta không cần quan tâm tới phần tử cuối cùng của exScan).

- **B3 : Tìm rank của digit.**

VD: với digit là  $[1\ 1\ 1\ 0\ 0\ 0]$  thì rank là  $[0\ 1\ 2\ 0\ 1\ 2]$ .

- **B4 : Scatter :  $\text{out}[\text{scan}[\text{digit}[i]] + \text{rank}[i]] = \text{in}[i]$ .**

VD: Sau khi scatter,  $A' = [2\ 6\ 4\ 1\ 3\ 5]$

### Kết quả thực thi: student\_func\_1.cu

	histogram's time	scan's time	scatter's time	total time
<b>tuần tự (k-bit = 1)</b>	12.1913	0.0010	17.0665	29.5771
<b>song song (ver 1)</b>	34.9571		52.6763	91.7025
<b>sort (in thrust)</b>				3.3823

Dễ thấy thời gian thực thi phiên bản song song thậm chí chậm hơn hẳn so với phiên bản tuần tự. Điều này đòi hỏi sự cải tiến phương pháp cài đặt để tăng tốc thời gian thực thi ít nhất là nhanh hơn phiên bản tuần tự trước khi so sánh với thuật toán sort trong thrust.

\*\*\*scatter's time bao gồm thời gian thực thi của b2, b3 và b4.

## Kết quả thực thi phiên bản 1:

```
phuonghoa@phuonghoa-X450LD: ~/Desktop/Problem Set 4
ed_eye_effect_template_5.jpg
hist: 34.954395 msecs
scatter: 52.489693 msecs

Your code ran in: 91.507553 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 34.955841 msecs
scatter: 52.962151 msecs

Your code ran in: 92.024925 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 34.962212 msecs
scatter: 52.597118 msecs

Your code ran in: 91.609756 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 34.956097 msecs
scatter: 52.577019 msecs

Your code ran in: 91.667709 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$
```

## 2.2. Phiên bản 2

### Phân tích phiên bản 1:

Có hai vấn đề đáng quan tâm ở phiên bản 1: histogram's time và scatter's time.

- Xét bước histogram dữ liệu: việc histogram trên toàn bộ dữ liệu nhưng số bin rất nhỏ ( $nBins = 2$ ) dẫn đến việc histogram không hiệu quả về mặt thời gian thực thi (vì với  $nBins = 2$  thì histogram kernel thực thi không khác mấy so với tuần tự.  
VD: với  $nElems = 1000$ , có 400 phần tử thuộc bin 0, 600 phần tử thuộc bin 1 thì histogram kernel sẽ thực hiện song song một cách tuần tự qua hàm `atomicAdd()`: tuần tự 400 lần đối với bin 0 và 600 lần với bin 1.  
Như vậy hàm histogram rõ ràng không hiệu quả về mặt thực thi song song.  
Vấn đề cần giải quyết ở đây là tăng số bin. Giải pháp dễ thấy nhất là tăng số digit trong một vòng lặp (tức  $nBits > 1$ ).

Nói tóm lại, để cải thiện tốc độ hàm histogram, việc trước nhất cần làm là tăng  $nBins$ , tức là tăng  $nBits$  (số digit trong một vòng lặp), vì  $nBins = 2^{nBits}$ .

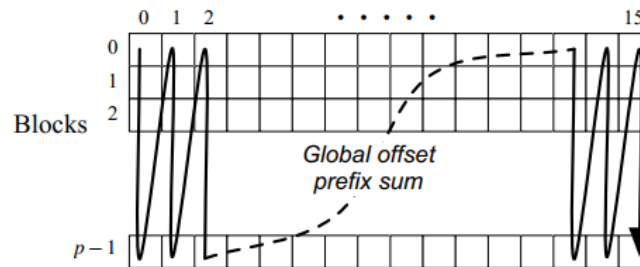
- Scatter's time là tổng thời gian của 3 bước: exclusive scan trên digit, tìm rank của digit và scatter.  
Nếu  $nBits$  tăng (do  $nBins$  tăng), việc exclusive scan trên digit rõ ràng không còn hợp lý nữa (vì bước exclusive scan trên digit và tìm rank dựa vào kết quả exclusive scan chỉ đúng khi  $nBits = 1$ ), điều đó đòi hỏi phải tìm một phương pháp khác giải quyết việc tính rank của  $nBits$ -digit.  
Phương pháp thay thế dễ thấy nhất (có thể không đảm bảo thời gian thực thi hiệu quả so với tính rank thông qua exclusive scan của digit) là tính số phần tử bằng  $a$  và đứng trước  $a$  trong mảng bằng một vòng lặp trong thread block.

Xét kỹ hơn vấn đề tăng nBits, rõ ràng nếu xử lý như nêu trên, tốc độ thực thi histogram kernel sẽ được cải thiện nhưng đổi lại thời gian thực thi với rank kernel thì chưa chắc (thậm chí có thể chậm hơn).

Điều này đòi hỏi giải pháp tăng nBits mà thời gian thực thi rank kernel ít nhất là không chậm hơn.

Tham khảo bài báo “*Designing Efficient Sorting Algorithms for Manycore GPUs*” - Nadathur Satish, Mark Harris và Michael Garland, mục Radix Sort thì ý tưởng giải quyết như sau:

- Phần histogram, thay vì chỉ bao gồm nBins thì cấu trúc của mảng histogram sẽ như sau:



Kích thước mảng histogram là  $nBins * p$  (với  $p$  là số block của histogram kernel).

Thực hiện histogram trên dữ liệu cục bộ trong mỗi block, tức là chỉ tìm số lần xuất hiện của nBits-digit trong phạm vi dữ liệu được phân cho một block.

Như vậy, cùng với việc tăng nBins, kích thước mảng histogram đã lớn hơn rất nhiều so với phiên bản 1, giúp cải thiện tốc độ thực thi. Đồng thời, việc thiết kế histogram như hình trên làm tiền đề để giải quyết việc tính rank hiệu quả.

- Phần tính rank của nBits-digit, trong mỗi thread block, thay vì chạy vòng lặp từ phần tử đầu tiên (chỉ số = 0) tới phần tử đứng trước liền kề nó (số vòng lặp có thể lên tới nElements => rất lớn), thì với cách bố trí histogram như trên, số vòng lặp lúc này gói gọn trong kích thước của một block. Điều này sẽ giúp tốc độ thực thi cải thiện đáng kể (ít nhất là không chậm hơn so với phiên bản 1).

Ở phần tính rank, có 2 cách tính:

- Cách 1: Trong mỗi block, mỗi thread phụ trách tính rank của một phần tử (thực hiện vòng lặp từ phần tử đầu tiên tới phần tử liền kề trước nó).
- Cách 2: Mỗi block (kích thước block chỉ là 1 threads) phụ trách tính rank của k phần tử (k là kích thước block của histogram kernel). Ở đây, bản thân mỗi block tính toán một cách tuần tự.

Trong phạm vi phiên bản 2, cả hai cách đều được cài đặt. Qua đo đạc thời gian, lựa chọn cách 1 làm mặc định.

Thiết kế: Thực hiện radix sort với  $nBits > 1$ .

- **B1:** Histogram trên digit ( $nBins = 2^i, i \geq 1$ ).

Histogram dữ liệu trên mỗi block (thay vì trên toàn bộ dữ liệu như phiên bản 1).

Kích thước của histogram là  $nBins * gridSize.x$  (với  $gridSize.x$  là số block của histogram kernel).

Trong đó, histogram sẽ là mảng 1 chiều (để phục vụ cho việc tính exclusive scan trên mảng histogram được hiệu quả hơn).

Xét một thread nhất định trong block thứ *blk*, với kích thước grid là *g*, nBits-digit thuộc bin *b*, thì vị trí tính histogram tương ứng của nó là *hist[b\*g+blk]*.

- **B2:** Exclusive scan trên histogram (tương tự phiên bản 1)
- **B3:** Tính rank của nBits-digit trong mỗi block dữ liệu.  
Do nBits > 1 nên việc tính rank sẽ khác so với phiên bản 1.  
Với mỗi phần tử trong một thread block, thực hiện vòng lặp đếm số phần tử bằng nó và đứng trước nó.
- **B4:** Scatter (tương tự phiên bản 1, cập nhật vị trí phù hợp với histogram trong phiên bản này).

Để đảm bảo đồng nhất dữ liệu xử lý, kích thước block và grid của histogram kernel, rank kernel và scatter kernel là như nhau.

#### Kết quả thực thi: student\_func\_2.cu

Kết quả phiên bản 2 nhanh hơn khoảng 3.86 lần so với phiên bản 1. Tuy nhiên, nếu so với thuật toán sort trong thư viện thrust thì vẫn chậm hơn rất nhiều.

	histogram's time	scan's time	scatter's time	total time
<b>tuần tự (k-bit = 8)</b>	1.3089	0.0012	3.7961	5.3708
<b>song song (ver 1)</b>	34.9571		52.6763	91.7025
<b>song song (ver 2)</b>	7.0347	4.6256	10.1447	23.7413
<b>sort (in thrust)</b>				3.3823

#### Kết quả thực thi phiên bản 2:

```

phuonghoa@phuonghoa-X450LD: ~/Desktop/Problem Set 4
./o -L /usr/lib -lopencv_core -lopencv_imgproc -lopencv_highgui -O3 -D_FORCE_INLI
NES -arch=sm_20 -Xcompiler -Wall -Xcompiler -Wextra -m64
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 7.040960 msecs
scan: 4.978304 msecs
scatter: 10.141439 msecs

Your code ran in: 24.101120 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 7.013952 msecs
scan: 4.439488 msecs
scatter: 10.139999 msecs

Your code ran in: 23.537216 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 7.049088 msecs
scan: 4.459008 msecs
scatter: 10.152736 msecs

Your code ran in: 23.585600 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$

```

### 2.3. Phiên bản 3 (cập nhật bổ sung phần thiết kế và thực thi)

#### Phân tích phiên bản 2:

Với việc áp dụng một phần cài đặt được mô tả trong bài báo “*Designing Efficient Sorting Algorithms for Manycore GPUs*” - Nadathur Satish, Mark Harris và Michael Garland, tốc độ ở phiên bản 2 rõ ràng đã cải thiện đáng kể so với phiên bản 1 (tăng gấp gần 4 lần).

Tiếp tục từ phiên bản 2, nhận xét thời gian thực thi ở 3 bước như sau:

- histogram’s time: việc áp dụng bố trí mảng histogram như trong phiên bản 2 về cơ bản đã giải quyết được việc thực thi song song trong histogram.
- scan’s time: tăng đáng kể so với phiên bản 1 (do tăng nBins và kích thước histogram lớn hơn nhiều), do đó không đáng ngại. Nói chung là chấp nhận được.
- scatter’s time: bao gồm cả thời gian tính rank.

Như đã phân tích ở phiên bản 2, việc tính rank thông qua thực hiện một vòng lặp trong thread block, mặc dù số lần lặp tối đa đã được hạn chế (nhờ cách bố trí mảng histogram) nhưng việc tính rank như thế này về cơ bản chưa hẳn là hiệu quả nhất. Nói cách khác có thể tìm một cách khác tính rank hiệu quả hơn.

Ở phiên bản này, vấn đề sẽ tập trung vào việc tìm cách tính rank hiệu quả hơn so với phiên bản 2. Tiếp tục áp dụng cách cài đặt triệt để hơn ở bài báo nêu trên, cách giải quyết được nêu ra như sau:

- Ở kernel histogram, thay vì chỉ thực hiện histogram, lúc này, trong mỗi block, load dữ liệu vào SMEM, tiến hành sắp xếp dữ liệu cục bộ trong block trên bộ nhớ chia sẻ và tính histogram trên dữ liệu đó, cuối cùng load dữ liệu xuống GMEM. Việc sắp xếp được thực hiện bằng phương pháp counting sort trên từng digit của mỗi nBits-digit.
- Ở rank kernel, số vòng lặp lúc này giới hạn từ phần tử  $a$  trở về trước tới khi gặp phần tử nhỏ hơn  $a$ .
- Nói cách khác, ở đây có một sự đánh đổi tốc độ histogram kernel với rank kernel. Câu hỏi là việc đánh đổi này có mang lại một kết quả khả quan hay không mà thôi.

Thiết kế: So với phiên bản 2, thực hiện bổ sung một thuật toán sort trong histogram kernel và thay đổi cách tính rank trong rank kernel.

- Sort and histogram kernel: áp dụng thuật toán counting sort để sắp xếp dữ liệu cục bộ của block đồng thời tìm histogram (quá trình tính toán thực hiện trên SMEM, sau đó load dữ liệu về lại GMEM).
- Rank kernel: Để tính rank của mỗi phần tử (trong 1 block), thực hiện vòng lặp từ phần tử  $a$  trở về trước cho đến khi gặp phần tử nhỏ hơn thì dừng.



### Kết quả thực thi: student\_func\_3.cu

Thời gian thực thi phiên bản 3 không nhanh như mong đợi, nó thậm chí còn chậm hơn phiên bản 2. Điều này cho thấy việc cải thiện không hiệu quả (phần cài đặt cần xem xét lại!!!)

	histogram's time	scan's time	scatter's time	total time
<b>tuần tự (k-bit = 8)</b>	1.3089	0.0012	3.7961	5.3708
<b>song song (ver 1)</b>	34.9571		52.6763	91.7025
<b>song song (ver 2)</b>	7.0347	4.6256	10.1447	23.7413
<b>song song (ver 3)</b>	11.5909	8.6928	8.5875	31.3346
<b>sort (in thrust)</b>				3.3823

### Kết quả thời gian thực thi:

```
phuonghoa@phuonghoa-X450LD: ~/Desktop/Problem Set 4
phuonghoa@phuonghoa-X450LD:~$ cd Desktop/'Problem Set 4'
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ make
nvcc -c student_func.cu -O3 -D_FORCE_INLINES -arch=sm_20 -Xcompiler -Wall -Xcomp
iler -Wextra -m64
nvcc -o HW4 main.o student_func.o HW4.o loadSaveImage.o compare.o reference_calc
.o -L /usr/lib -lopencv_core -lopencv_imgproc -lopencv_highgui -O3 -D_FORCE_INLI
NES -arch=sm_20 -Xcompiler -Wall -Xcompiler -Wextra -m64
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 11.611584 msecs
scan: 8.861888 msecs
scatter: 8.591969 msecs
Your code ran in: 31.521952 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$ ./HW4 red_eye_effect_5.jpg r
ed_eye_effect_template_5.jpg
hist: 11.570176 msecs
scan: 8.523616 msecs
scatter: 8.583040 msecs
Your code ran in: 31.147297 msecs.
phuonghoa@phuonghoa-X450LD:~/Desktop/Problem Set 4$
```

## 2.4. Phiên bản 4 (bổ sung – chưa cài đặt được)

Ý tưởng phiên bản 4: tiếp tục với cách phân tích từ phiên bản 3, tuy nhiên cải tiến phần cài đặt counting sort trong bước histogram và phần tính rank.

- Counting sort: thay vì cài đặt counting sort một cách thuần túy như phiên bản 3 (ở phiên bản 3, vị trí một phần tử được xác định bằng vòng lặp tìm số phần tử đứng trước nhỏ hơn hoặc bằng nó và một vòng lặp khác tìm số phần tử đứng sau nhỏ hơn nó). Trong phiên bản này, thực hiện counting sort trên từng digit.  
Cụ thể: với mỗi k-bit, thực hiện k vòng lặp, mỗi vòng lặp thực hiện counting sort trên dữ liệu nhị phân (thực hiện exclusive scan, sau đó tìm rank của phần tử tương ứng).
- Rank (trên dữ liệu đã sort): tính exclusive scan trên histogram cục bộ của mỗi block  
=> tìm rank dựa vào kết quả scan và vị trí hiện tại của nó.  
VD: A = [0 1 3 0 2 3 1 0] được phân vào 2 block như sau:  
**block 0:** 0, 1, 3, 0      **block 1:** 2, 3, 1, 0  
Sau khi sắp xếp:  
**block 0:** 0, 0, 1, 3      **block 1:** 0, 1, 2, 3



Histogram trên mỗi block

block\bin	0	1	2	3
0	2	1	0	1
1	1	1	1	1

Scan trên histogram cục bộ của block (khác với scan trên toàn bộ dữ liệu histogram ở bước 2).

block\bin	0	1	2	3
0	0	2	3	3
1	0	1	2	3

Rank của từng phần tử trên mỗi block:  $\text{rank}(a[i]) = i - \text{scan}[a[i]]$

Vd: block 0:  $a[0] = 0, \text{rank}(a[0]) = 0 - \text{scan}[0] = 0 - 0 = 0$   
 $a[1] = 0, \text{rank}(a[1]) = 1 - \text{scan}[0] = 1 - 0 = 1$   
 $a[2] = 1, \text{rank}(a[2]) = 2 - \text{scan}[1] = 2 - 2 = 0$   
 $a[3] = 3, \text{rank}(a[3]) = 3 - \text{scan}[3] = 3 - 3 = 0$

block 1: tương tự như trên.

Phiên bản 4 được dự đoán sẽ cải thiện thời gian thực thi so với phiên bản 2 và 3 do tối ưu hóa những phép tính không cần thiết.

## 2.5. Phiên bản tốt nhất

Trong phạm vi bài báo cáo này, phiên bản thực thi song song tốt nhất là phiên bản 2. Ở phiên bản 2, có 5 kernel. Tuy nhiên, histogram kernel, rank kernel và scatter kernel có cùng kích thước block nên chỉ có ba loại kích thước block là blockDigit, blockSize (cho 3 kernel đã nêu) và blockScan.

Thông kê thời gian thực thi trên các kích thước block khác nhau như sau (đơn vị: ms)

blockDigit	blockSize	blockScan	time
32	32	32	71.2215
64	32	32	70.6016
128	32	32	70.4096
256	32	32	70.3896
512	32	32	70.4056
256	64	32	42.6695
256	128	32	30.8190
256	256	32	26.4750
256	512	32	26.9885
256	1024	32	37.9841
256	256	64	24.7840
256	256	128	23.8802
256	256	256	23.6013



### 3. Quá trình làm việc nhóm

#### 3.1. Phiên bản tuần tự

Mỗi thành viên tự nghiên cứu và viết một phiên bản tuần tự riêng => thống nhất bàn bạc để lựa chọn phiên bản tốt hơn.

#### 3.2. Phiên bản song song

- Phiên bản 1: bắt đầu từ trường hợp đơn giản nhất ( $n\text{Bits} = 1$ ):  
Thành viên 1: cài scan kernel  
Thành viên 2: cài các kernel còn lại  
Phần cài đặt trong hàm `your_sort()`: mỗi thành viên tự tổng hợp kernel và cài đặt riêng  
=> chọn phiên bản tốt hơn.
- Phiên bản 2:  
Nghiên cứu bài báo: tự đọc  
Kế hoạch phát triển phiên bản 2: ý kiến thống nhất từ 2 thành viên  
Phần cài đặt phiên bản 2: giữ nguyên scan kernel  
Thành viên 2: cập nhật các kernel còn lại  
Thành viên 1: tổng hợp kernel và cài đặt trong hàm `your_sort()`
- Phiên bản 3: Mỗi thành viên tự cài đặt riêng hàm histogram kernel (có counting sort) và rank kernel.