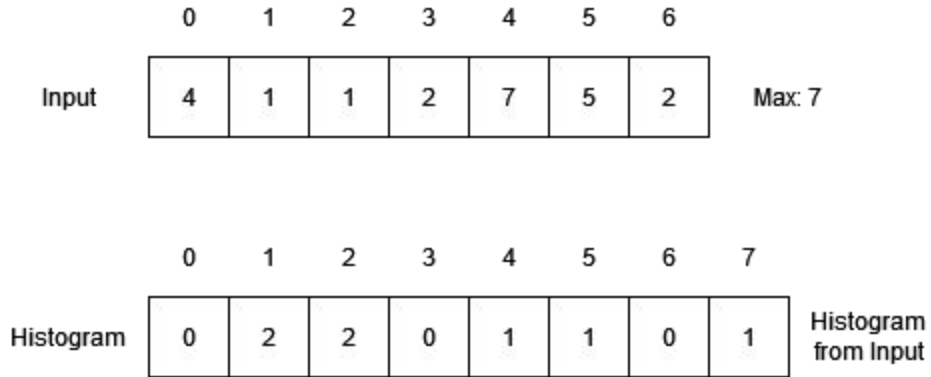


# CUDA LSD Radix Sort

# Counting Sort: Histogram



NOTE: The histogram size depends on the maximum value in the input

# Counting Sort: Offsets

	0	1	2	3	4	5	6	7	
Histogram	0	2	2	0	1	1	0	1	Histogram from Input
Offsets	0	0	2	4	4	5	6	6	Exclusive Prefix Sum on Histogram

# Counting Sort: Scatter

```
for each elem in Input:
```

```
    d = Offsets[elem]++
```

```
    Output[d] = elem
```

NOTE: Counting Sort is Stable!

	0	1	2	3	4	5	6
Input	4	1	1	2	7	5	2

	0	1	2	3	4	5	6	7
Offsets	0	2	4	4	5	6	6	7

	0	1	2	3	4	5	6
Output	1	1	2	2	4	5	7

# Least Significant Digit (LSD) Radix Sort

The idea of LSD Radix Sort is to repeatedly perform counting sort using as key the  $i$ -th digit, going from least significant to most significant. This works because Counting Sort is stable

NOTE: Using digits as keys, histograms are smaller

	0	1	2	3	4	5	6	7	8	9
Input	11	131	742	9	66	122	634	93	5	873

11	131	742	9	66	122	634	93	5	873
----	-----	-----	---	----	-----	-----	----	---	-----

Pass 0

11	131	742	122	93	873	634	05	66	09
----	-----	-----	-----	----	-----	-----	----	----	----

Pass 1

005	009	011	122	131	634	742	066	873	093
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Pass 2

Output	005	009	011	066	093	122	131	634	742	873
--------	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

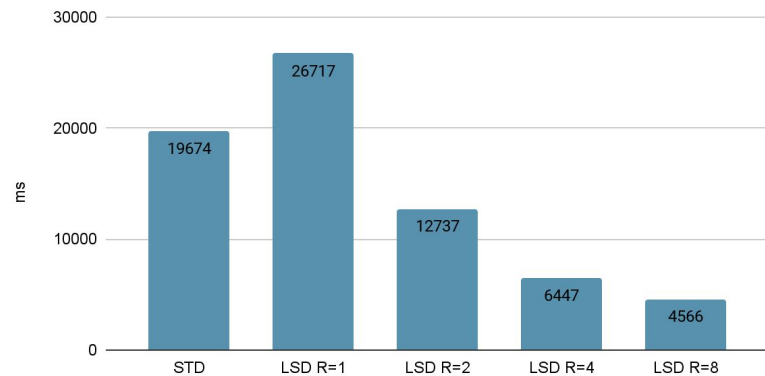
# Implementation Details

- We sort arrays of 32 bit unsigned integers
- We sort by groups of **R** bits, from least to most significant
  - We support the following values for R: 1, 2, 4, 8
- We expect our input to have a number of elements that is a power of 2
- We expect our blocks to have dimensions which are a power of 2

# LSD Radix Sort vs STD Sort (Comparison Based) CPU

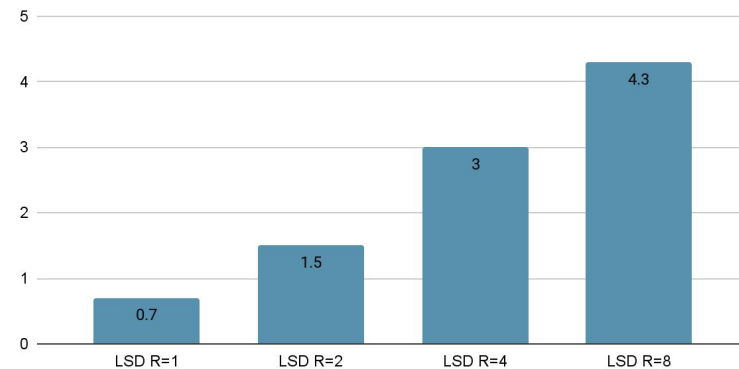
Time

1GB of data






Speedup

1GB of data



# CUDA LSD Radix Sort

We have seen that the key component to LSD Radix Sort is Counting Sort. We must find a way to parallelize Counting Sort. We can observe that

1. The computation of a histogram is trivially parallelizable 
2. The computation of a prefix sum is a form of scan, and as such, it is easy to parallelize 
3. The scatter phase is strictly sequential, here lies the problem 

Parallelizing LSD Radix Sort is not trivial. We need a novel approach

Harada, T., & Howes, L.W. (2011). Introduction to GPU Radix Sort.



# CUDA LSD Radix Sort Pass

Given arrays **A** (input) and **B** (output) of size **s** and block dimension **bdim**

1. **Build Histograms.** We split **A** into  $p = s / \text{bdim}$  blocks. Every block **b** has the task of building an histogram **H<sub>b</sub>** ( $2^R$  buckets) for its input elements. **H<sub>1</sub>, H<sub>2</sub>, ..., H<sub>p</sub>** are laid sequentially into **H**
2. **Build Local and Global Offsets.** At first we copy **H** into **C**
  - a. Local Offsets **L**. Every block **b** performs an exclusive prefix sum on **H<sub>b</sub>**
  - b. Global Offsets **G**. We consider **C** as a  $p \times 2^R$  matrix. We transpose **C** into **C<sup>T</sup>**. We perform an exclusive prefix sum on **C<sup>T</sup>**. Finally we transpose again **C<sup>T</sup>**
3. **Scatter.**
  - a. Every block **b** sorts in place its input elements
  - b. Given a thread with index **i** and its input element **n** with key **k**
  - c. We compute the destination index **d** of **n** as:  $d = i - L[b][key] + G[b][key]$
  - d. We write **B[d] = n**

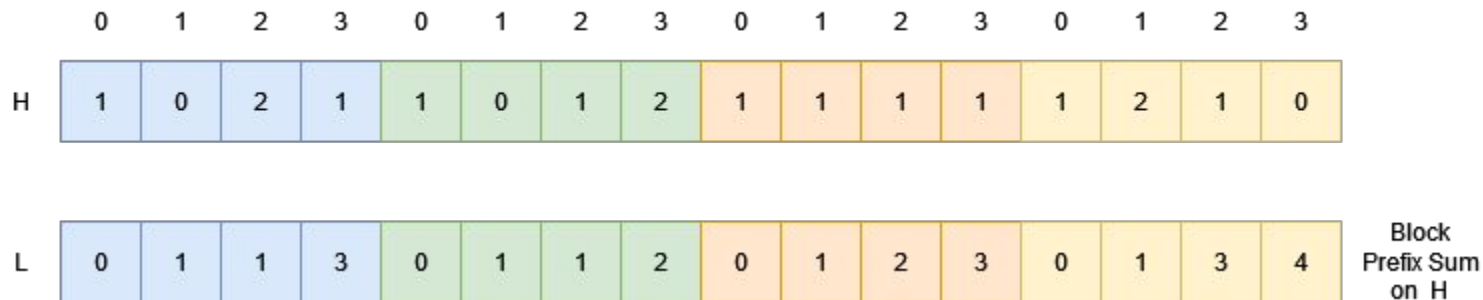
# CUDA LSD Radix Sort Pass Example: Build Histograms

$s = 16$ ,  $\text{bdim} = 4$ ,  $R = 2$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0	3	2	2	3	2	0	3	2	1	0	3	2	0	1	1
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
H	1	0	2	1	1	0	1	2	1	1	1	1	1	2	1	0

# CUDA LSD Radix Sort Pass Example: Build Local Offsets

$s = 16$ ,  $\text{bdim} = 4$ ,  $R = 2$



# CUDA LSD Radix Sort Pass Example: Build Global Offsets

	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	
H	1	0	2	1	1	0	1	2	1	1	1	1	1	2	1	0	
C	1	0	2	1	1	0	1	2	1	1	1	1	1	2	1	0	
C <sup>T</sup>	1	1	1	1	0	0	1	2	2	1	1	1	1	2	1	0	
C <sup>T</sup>	0	1	2	3	4	4	4	5	7	9	10	11	12	13	15	16	Prefix Sum on C <sup>T</sup>
G	0	4	7	12	1	4	9	13	2	4	10	15	3	5	11	16	Transpose C <sup>T</sup>

# CUDA LSD Radix Sort Pass Example: Scatter

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0	3	2	2	3	2	0	3	2	1	0	3	2	0	1	1
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
A	0	2	2	3	0	2	3	3	0	1	2	3	0	1	1	2
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
L	0	1	1	3	0	1	1	2	0	1	2	3	0	1	3	4
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
G	0	4	7	12	1	4	9	13	2	4	10	15	3	5	11	16
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
d	0	7	8	12	1	9	13	14	2	4	10	15	3	5	6	11
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	0	0	0	0	1	1	1	2	2	2	2	2	3	3	3	3

In place block sort

Scatter

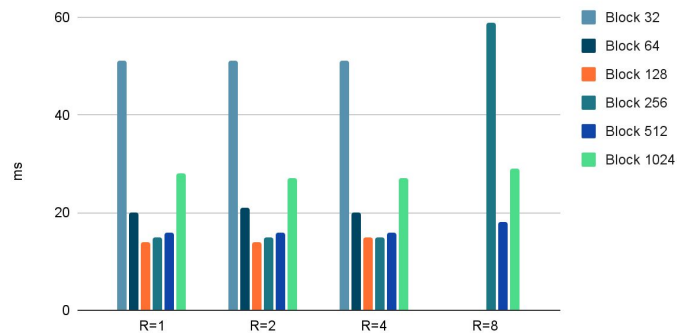
# Build Histograms: Implementation Details

- Split input by block
- Each block builds an histogram of its own input
- The block builds the histogram in **shared memory**
- We do this to avoid **random global memory accesses**
- We may have some **bank conflicts** (this is still better than random global memory writes)
- At the start we must **zero initialize** the **shared memory**
- To avoid **race conditions** we use **atomic operations** when we update the histogram
- At the end, the block copies the histogram from shared memory to global memory, resulting in **aligned and coalescent global memory writes**

# Build Histograms: Performance

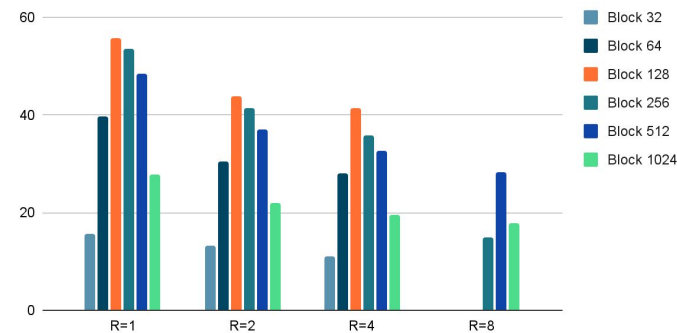
## GPU Time

4GB of data - CPU took 800 ms



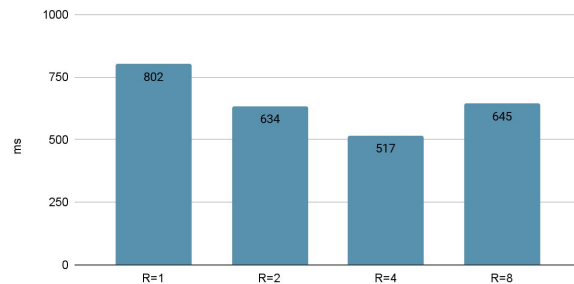
## Speedup

4GB of data



## CPU Time

4GB of data



# Parallel Prefix Sum

Given an input array **A** of size **s** and block dimension **bdim**

- If  $s \leq \text{bdim}$  then we perform a block local prefix sum on **A**
- Otherwise we split **A** into  $p = s / \text{bdim}$  blocks
- Every block **b** performs a block prefix sum on its input
- Every block **b** has also the task of computing the block sum **Zb**. **Z1, Z2, ..., Zp** are laid sequentially into **Z**
- We perform a parallel prefix sum on **Z**
- We concurrently add **Z[i]** to each element of block **i + 1**

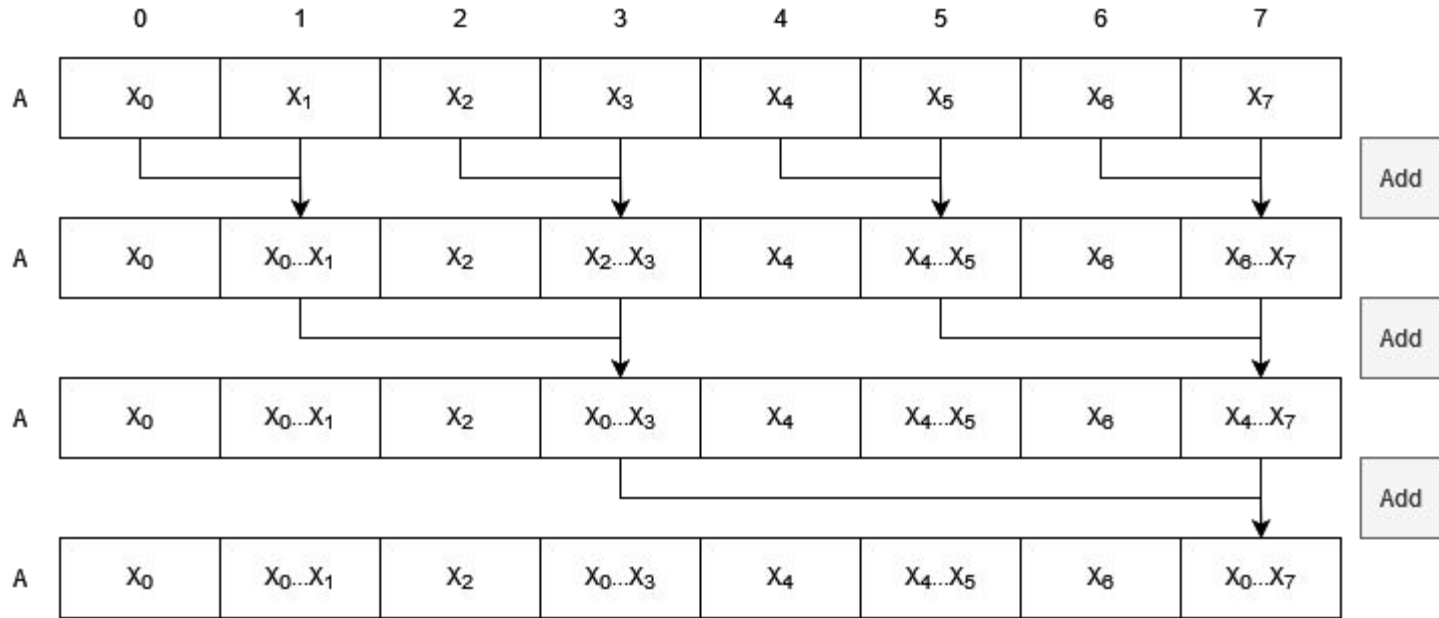
Nguyen, H. (2007). GPU Gems 3.



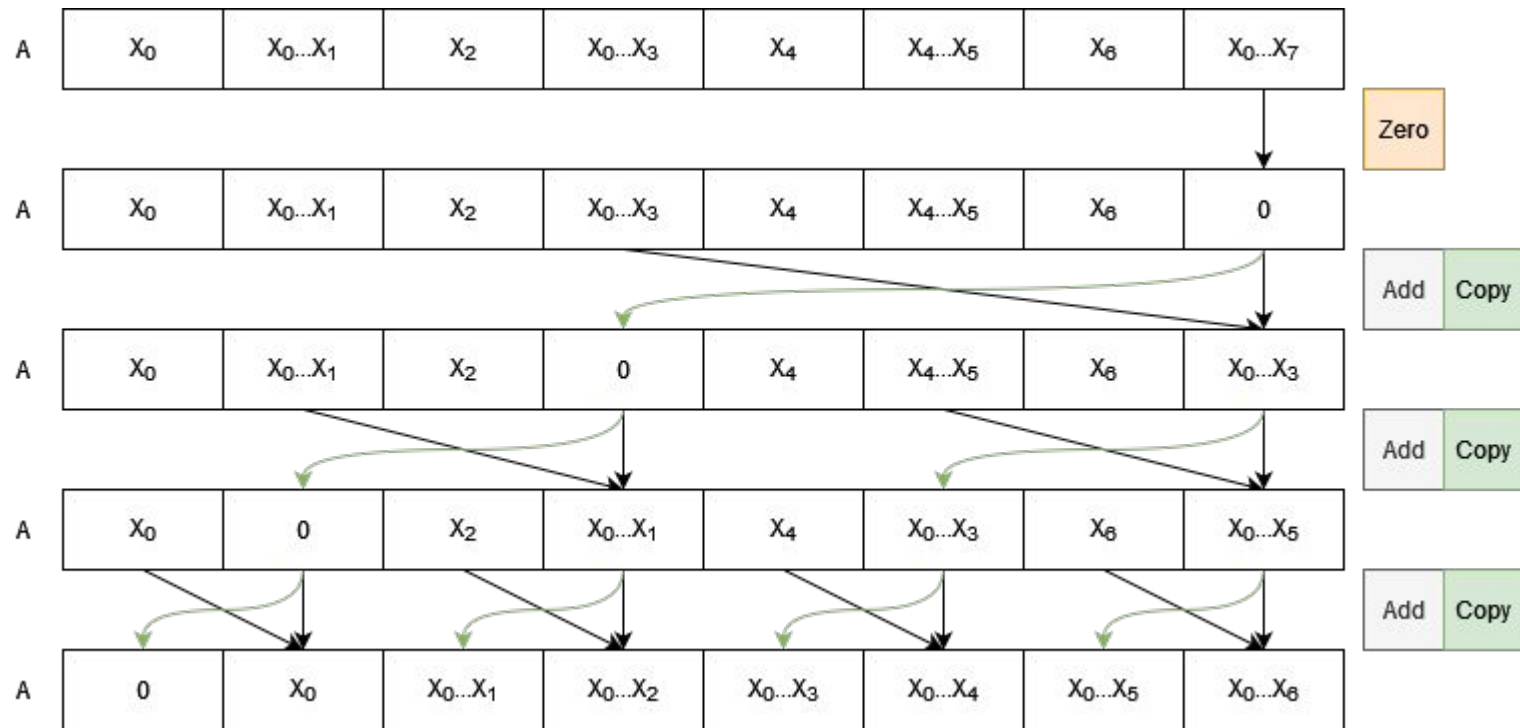
# Block Local Prefix Sum

- Let  $A$  be an input array
- We conceptually consider  $A$  to be a tree
- **Up Sweep.** We traverse the tree from leaves to root, computing partial sums at internal nodes. At the end, the root node (the last element of the array) holds the sum of all the elements
- **Down Sweep.** We insert zero at the root. We traverse the tree from root to leaves, building the scan in place. At every iteration, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child

# Block Local Prefix Sum: Up Sweep



# Block Local Prefix Sum: Down Sweep



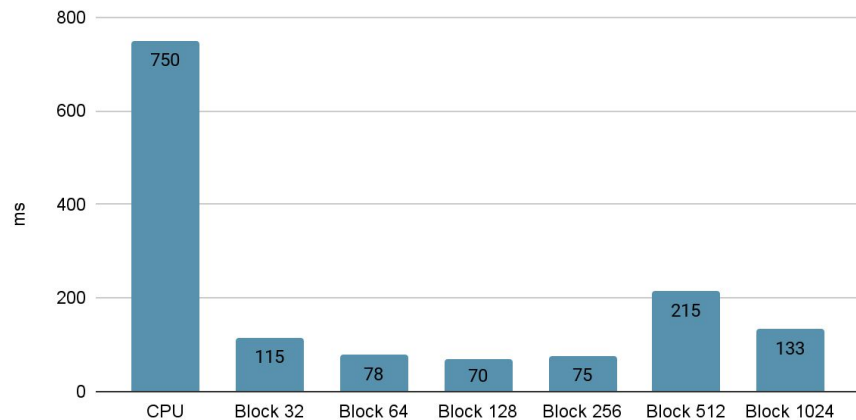
# Parallel Prefix Sum: Implementation Details

- In our Block Local Prefix Sum, we load our input data into **shared memory**
- We perform the Up Sweep and Down Sweep phases on **shared memory**
  - This suffers from bank conflicts but it is still better than directly using global memory
- Finally we write the **shared memory** back to **global memory**, resulting in an **aligned and coalesced global memory write**
- Before performing the Parallel Prefix Sum, we determine the **total number of block sums** that we will be computing and preallocate an appropriate array in device memory
- The parallel add is trivial

# Parallel Prefix Sum: Performance

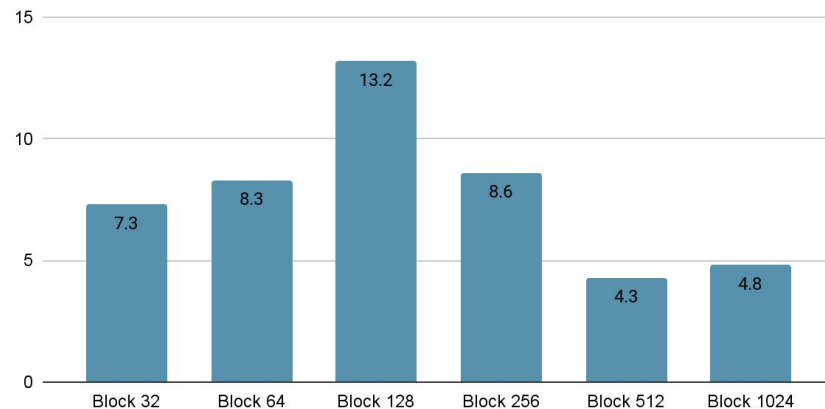
## GPU Time

4GB of data



## Speedup

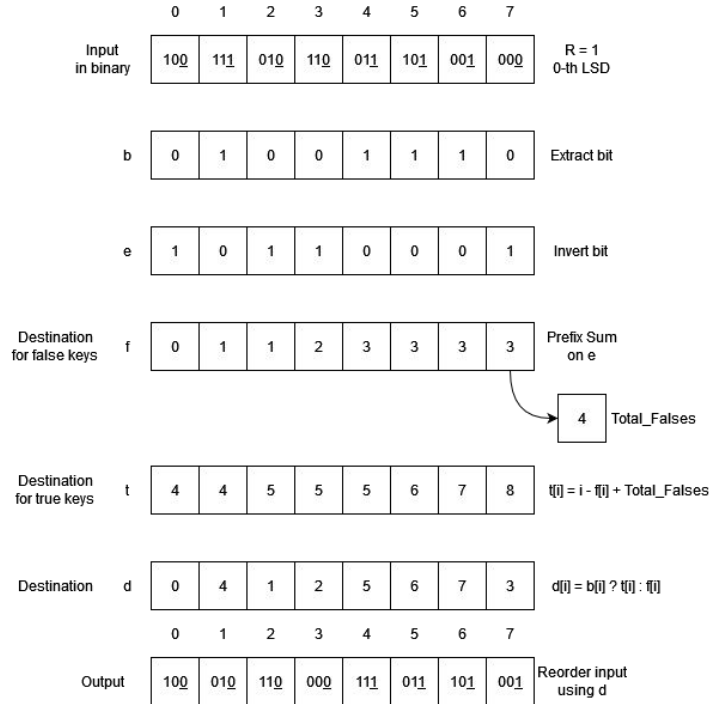
4GB of data



# In Place Sort - LSD Binary Radix Sort

Single pass example

Nguyen, H. (2007).  
GPU Gems 3.



# LSD Binary Radix Sort: Implementation Details

- We load our input array into **shared memory**
- We perform **R passes**
- To restrict our **shared memory** usage
  - Each thread saves its current input value in a **register**
  - Each thread writes its inverted bit into the **same shared memory** we used to load our input
  - We perform our **block local prefix sum** on the **shared memory**
  - Each thread computes **t** using a **register**
  - Each thread computes **d** using a **register**
  - Each thread writes back in **shared memory** its input value depending on **d**

# Scatter: Implementation Details

- We load our input, local offsets and global offsets into **shared memory**
- We use **LSD Binary Radix Sort** to sort in place our input elements loaded into **shared memory**. We **don't sort by value**. We **sort by key** (R passes)
- We compute the destination index using a **register**
- We scatter the elements into the output array, in **global memory**
- Our **global memory writes** may **not be aligned**
- In a block, all input elements with the **same keys** are written in a **coalesced** way
- In a block, input elements with **different keys** cause **random global memory writes**



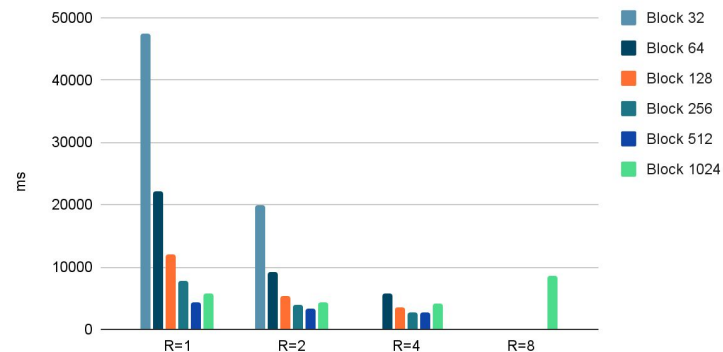
# CUDA LSD Radix Sort: Implementation Details

- We perform **32 / R passes**
- All kernels are launched on the **default stream**
- Since building the local and global offsets are **independent tasks**, we submit the kernels on **two different non default streams**
  - At the start we need a **copy** of our histograms
- When building the global offsets, we use a **fast transpose kernel** that uses shared memory to improve **global memory accesses**
  - <https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/>

# CUDA LSD Radix Sort: Performance

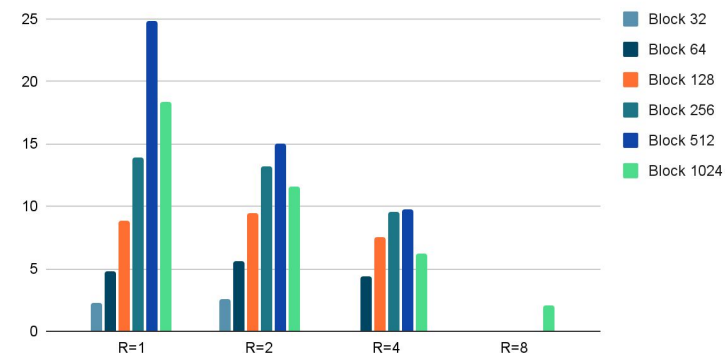
GPU Time

4GB of data



Speedup

4GB of data



CPU Time

4GB of data

