Distributed and Parallel Computing

Lecture 08

Alan P. Sexton

University of Birmingham

Spring 2019

Parallel Sorting

There are many serial algorithms that do not parallelize well. We want algorithms with:

- All many threads to work together on the problem
  - Serial algorithms are often inherently sequential
- Minimize branch divergence
  - Serial algorithms tend to do a lot of branching
- Coalesce memory access
  - Serial algorithms tend to access memory very randomly

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,. . .) with their neighbours to the right (1,3,5,. . .) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,. . .) with their neighbours to the right (1,3,5,. . .) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4
1	2	3	4	5

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4
1	2	3	4	5

- n inputs, steps: $O(n)$, work: $O(n^2)$

Parallel Merge Sort

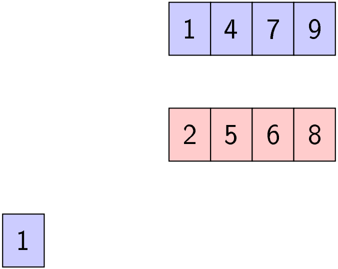
In the simplest form, Parallel Merge Sort works as follows:

- Start with a set of (trivially sorted) sequences of length 1
    - i.e. single elements
- In each step, merge independent pairs of sequences from the set of sorted sequences together to make a set of half the number of longer sorted sequences
- Finish when the last pair of sequences is merged into one final sorted sequence

Merging 2 sequences

Sequentially merging 2 sequences:

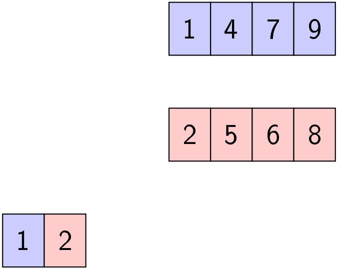
while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output

1	4	7	9

2	5	6	8

1	2	4

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output

1	4	7	9

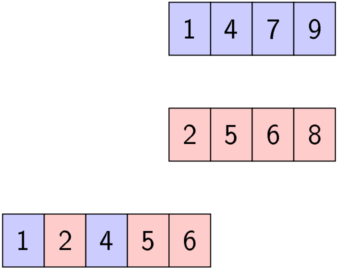
2	5	6	8

1	2	4	5

Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output

1	4	7	9

2	5	6	8

1	2	4	5	6	7

Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output

1	4	7	9

2	5	6	8

1	2	4	5	6	7	8

Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
  Compare the elements at the head of the 2 sequences
  Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output

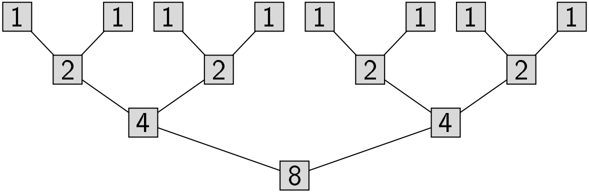
1	4	7	9

2	5	6	8

1	2	4	5	6	7	8	9

Parallel Merge Sort

Progressive sequence sizes shown:



- n inputs
- steps: $O(\log n)$
- work: $O(n \log n)$
  - In each step we are generating n elements.
  - Each element generated (except the last in each merge) is the result of one comparison
  - $n(1 - \frac{1}{2}) + n(1 - \frac{1}{4}) + n(1 - \frac{1}{8}) + \ldots$ with $\log n$ terms
  - $= n \log n - (n - 1)$
  - $= O(n \log n)$

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges
        - 1 thread: 1 merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges
        - 1 thread: 1 merge
    - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges
        - 1 thread: 1 merge
    - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges
        - 1 thread: 1 merge
    - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
    - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
    - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
    - Here there are many small merges to do: each thread can do one merge, each block handles many merges
        - 1 thread: 1 merge
    - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
    - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
    - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
        - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge ⇒ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs
   - Use multiple blocks to handle each merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs
   - Use multiple blocks to handle each merge
     - Multiple blocks of threads: 1 merge

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element

Merge: 1 Block of Threads to 1 Merge

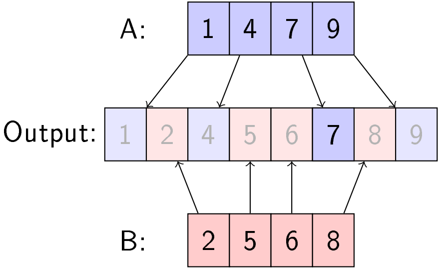
Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

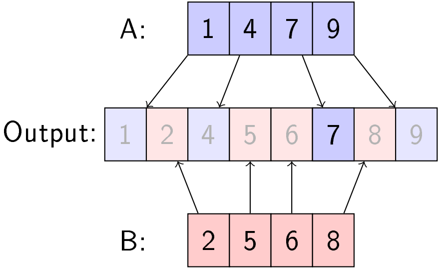
- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:



Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:

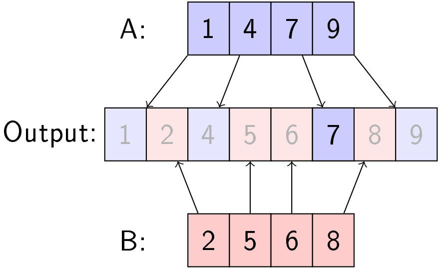


- Thread for $A[2]$ knows location in A is 2

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:

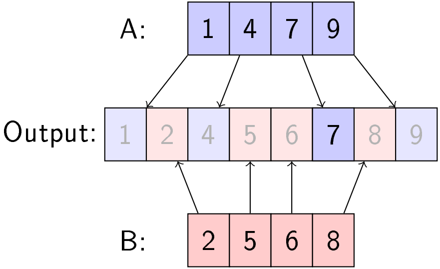


- Thread for $A[2]$ knows location in A is 2
- Thread does binary search to find insertion location in B is 3

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:



- Thread for $A[2]$ knows location in A is 2
- Thread does binary search to find insertion location in B is 3
- Hence location in output is $2 + 3 = 5$

Merge: 1 Block of Threads to 1 Merge

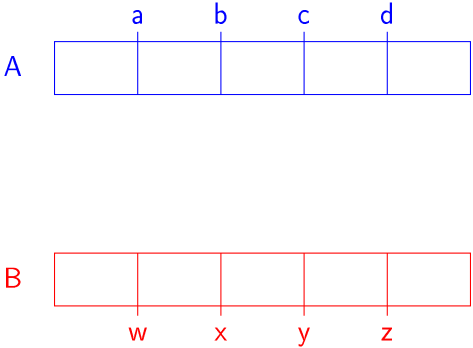
- Merge in a sequence of kernels
- Blocks per Grid is the number of merges to execute
- Threads per block is the number of elements that the merge will produce
- Copy sequences from global to shared memory, merge and copy back
- Thus (on GTX960s) suitable for merges that produce sequences of length 64 to 1024
  - GTX960 allows up to 32 blocks per SM, but can manage 2048 threads per SM. So less than 64 threads per block and the SM will not be fully occupied
- Can handle merges larger than 1024 elements:
  - Read chunks of sequences from global to shared memory, merge chunks and copy back
  - Slightly tricky to handle the *streams* of chunks

Merge: Multiple Blocks of Threads to 1 Merge

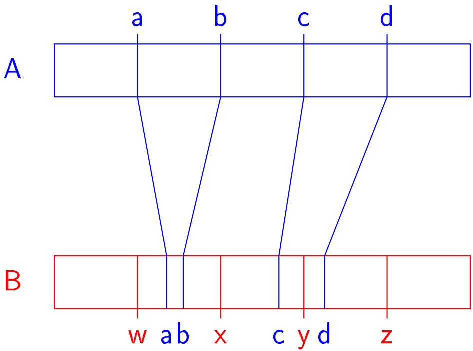
Problem is to break up a large merge so that different blocks can work on different parts of the merge independently

- Choose *splitters*, max K elements apart, from both sequences
- Merge the splitters into a single sorted list, remembering their locations in their home sequences
  - our previous medium merge method can do this
- Find the insertion location of each splitter in its *foreign* sequence (binary search)
  - Each splitter now has locations for both sequences
- Each consecutive pair of splitters thus defines a section of both sequences that can be merged independently of any other sections
- None of these sections can merge into more than $2K$ elements
- Choose K to be maximum 512 and each merge section can be handled by 1 block of 1024 threads

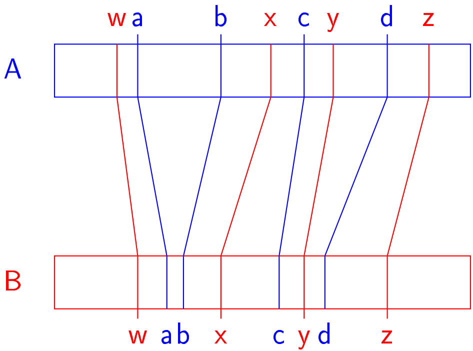
Merge: Multiple Blocks of Threads to 1 Merge



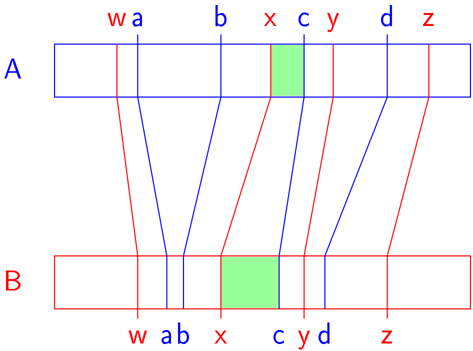
Merge: Multiple Blocks of Threads to 1 Merge



Merge: Multiple Blocks of Threads to 1 Merge



Merge: Multiple Blocks of Threads to 1 Merge

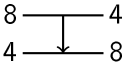


- $|[b, c]|$ in A is $K \Rightarrow |[x, c]| \leq K$ in A
- Similarly $|[x, c]| \leq K$ in B $+ \rightarrow \leq 2K$
- Hence the merge of the $[x, c]$ segments is no more than $2K$
- Similarly for all other segment pairs

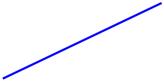
Bitonic Sort

Some definitions:

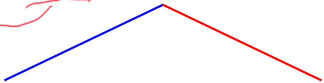
- A comparator is a function that swaps two elements if they are in the wrong order



- A monotonic increasing/decreasing sequence is one where every element is equal to or greater/less than every preceding element in the sequence
  - 1, 4, 8, 16, 16, 18, 19, 22



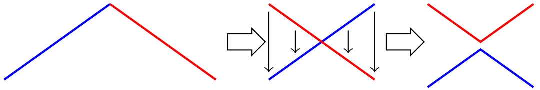
- A bitonic sequence is a sequence which changes order direction at most once, or a circular shift of such a sequence
  - 15, 12, 5, 2, 1, 4, 8, 16
  - 1, 4, 8, 16, 15, 12, 5, 2



Bitonic Split

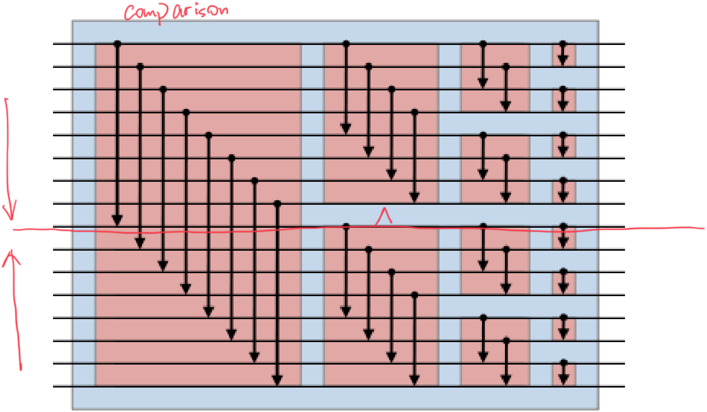
The central idea in Bitonic sort is that:

- A simple parallel arrangement of comparators can split a bitonic sequence into two bitonic sequences, where all elements of the first are less than all elements of the second:



									15	12	5	2		1	4	5	2	smaller
1	4	8	16	15	12	5	2	$\Rightarrow$					$\Rightarrow$					
									1	4	8	16		15	12	8	16	larger

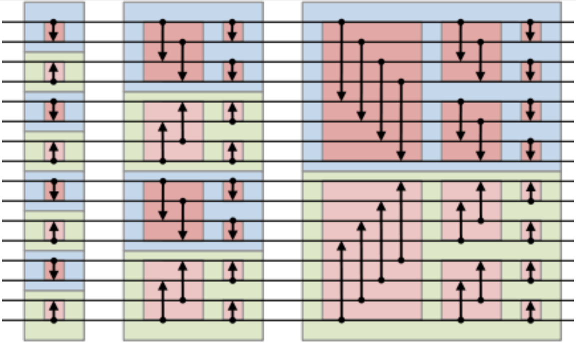
Bitonic Sort: Second Phase



- If the inputs along the left are a bitonic sequence:
  - First red block splits it into two bitonic sequences, where all upper half elements are less than all lower half ones
  - The next two red blocks splits these 2 into 4 similarly, etc.
  - Final output is sorted

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

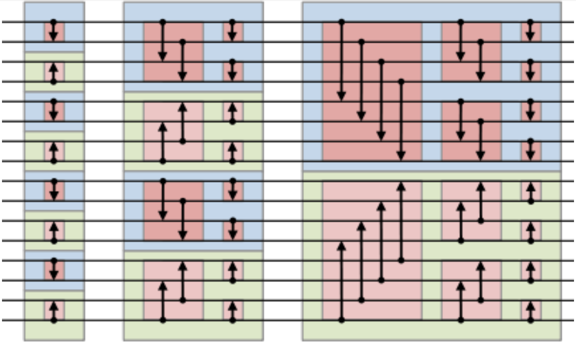
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

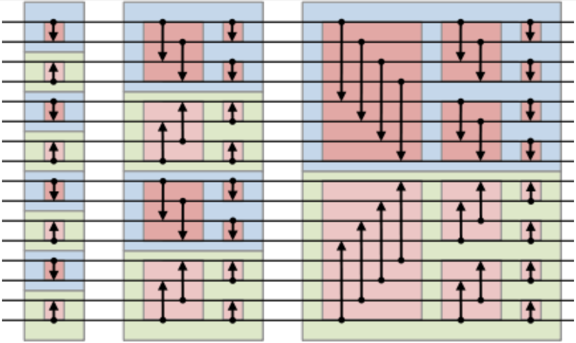
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

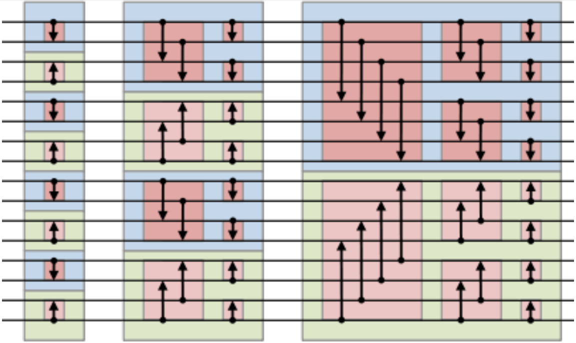
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

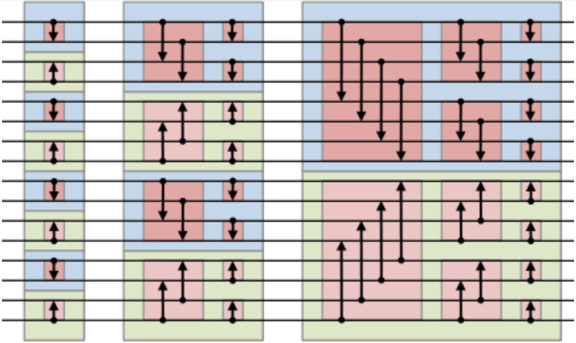
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16
  - Middle column turns 4 of len 4 $\rightarrow$ 2 of len 8

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

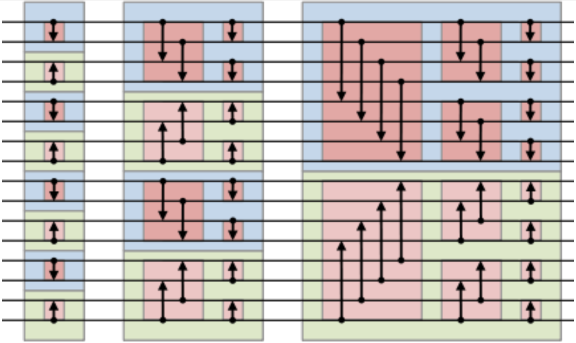
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\to$ 1 of len 16
  - Middle column turns 4 of len 4 $\to$ 2 of len 8
  - Left column turns 8 of len 2 $\to$ 4 of len 4

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

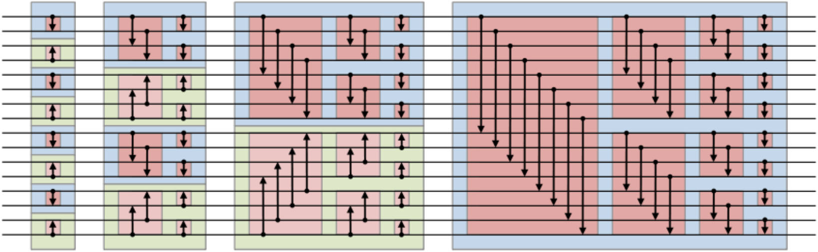
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16
  - Middle column turns 4 of len 4 $\rightarrow$ 2 of len 8
  - Left column turns 8 of len 2 $\rightarrow$ 4 of len 4
  - But all sequences of length 2 are trivially bitonic!

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

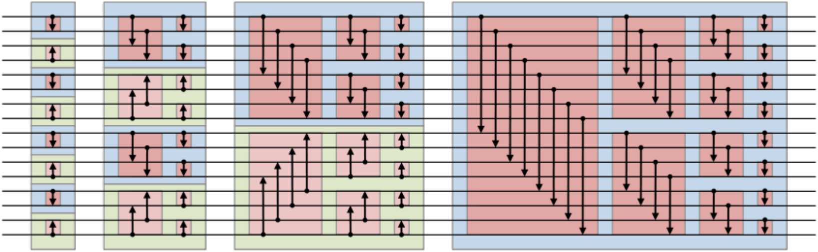
Bitonic Sort



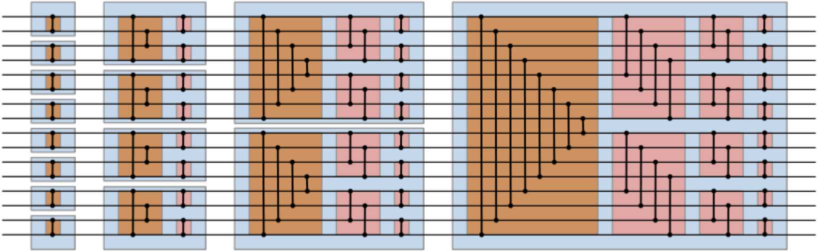
- Each column of red blocks runs in parallel with no races
- Assign each thread to one data element (Some implementations: 1 thread to one comparison)
- Each comparison executed twice:
  - At lower end, thread stores the smaller of the two values
  - At Upper end, thread stores the larger of the two values
- Complexity: $O(n \log^2 n)$ steps: but fastest sort for small sets
- Excellent for first stage of merge sort

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

Bitonic Sort



Can be rearranged with all arrows down:



images from https://en.wikipedia.org/wiki/Bitonic_sorter

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix} 0 \\ 5 \\ 2 \\ 7 \\ 1 \\ 3 \\ 6 \\ 4 \end{bmatrix} = \begin{bmatrix} 000 \\ 101 \\ 010 \\ 111 \\ 001 \\ 011 \\ 110 \\ 100 \end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix} 0 \\ 5 \\ 2 \\ 7 \\ 1 \\ 3 \\ 6 \\ 4 \end{bmatrix} = \begin{bmatrix} 000 \\ 101 \\ 010 \\ 111 \\ 001 \\ 011 \\ 110 \\ 100 \end{bmatrix} \rightarrow \begin{bmatrix} 000 \\ 010 \\ 110 \\ 100 \\ 101 \\ 111 \\ 001 \\ 011 \end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \to \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \to \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \to \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix}=\begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix}\rightarrow\begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix}\rightarrow\begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix}\rightarrow\begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix}=\begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \to \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \to \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \to \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements
- Reduce k: do 2^m splits by splitting on m bits at a time, e.g. 4

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements
- Reduce k: do 2^m splits by splitting on m bits at a time, e.g. 4
- Fastest CUDA GPU sort for medium to large inputs

Radix Sort Implementation

- Each split section can be generated with a compact operation:
  - Map on LSB = 0, followed by an exclusive sum scan to calculate the first section scatter addresses

    least significant bit.
  - Use the last scatter address calculated as an offset to the scatter addresses for the second section
  - If using multi-bit radix steps, run a histogram to calculate the number in each section and hence the offsets