GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures

Alexander Greß

Institute of Computer Science II
University of Bonn

Gabriel Zachmann

Institute of Computer Science
Clausthal University of Technology

universitätbonn

TU Clausthal

Contribution

IPDPS 2006 GREECE
International Parallel & Distributed Processing Symposium

A sorting algorithm for *stream processing architectures*,

- which has optimal time complexity O(n log n / p), in contrast to previous sorting approaches on stream architectures;
- especially suited for implementation on graphics hardware (GPUs);
- the optimized GPU implementation outperforms quick-sort on CPU as well as recent (non-optimal) sorting approaches on GPUs already for sequence sizes >= 32768.

Our approach is based on the PRAM sorting algorithm *Adaptive Bitonic Sorting* (Bilardi, Nicolau 1989).

universitätbonn TU Clausthal

Overview

- Background on stream architectures (and GPUs)
- Recent work on sorting on stream architectures
- Background on adaptive bitonic sorting
- First step towards a stream program
- Actual stream program (no random access writes)
- Implementation issues
- Results / Timings

universitätbonn TU Clausthal

Background: Stream architectures

Stream Programming Model:

„Streams of data passing through computation kernels.“

- *Stream*:
  Ordered set of data of arbitrary datatype.
- *Kernel*:
  Specifies the computation to be performed on *each* element of the input stream.

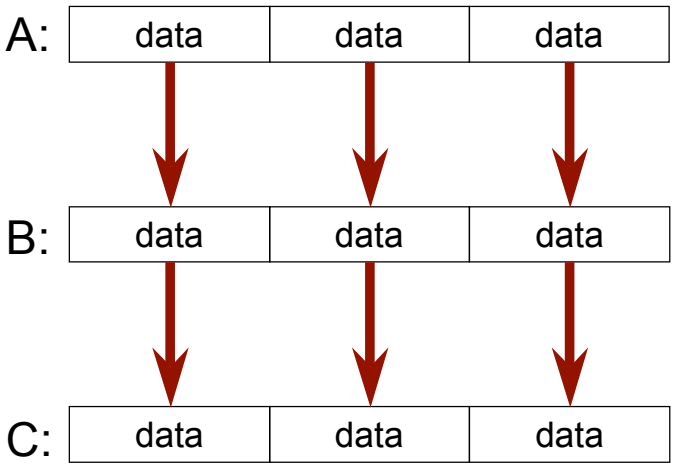
universitätbonn TU Clausthal

Background: Stream architectures

Sample stream program:

```
{
  stream A, B, C;
  ...
  kernelfunc1(input: A,
              output: B);

  kernelfunc2(input: B,
              output: C);
  ...
}
```



Background: Stream architectures

- Stream processor prototypes:
  Imagine, Merrimac, ...

- Programmable graphics hardware (GPUs):
  - Although originally built for graphics rendering, nowadays similar capabilities to stream processors.
  - Current trend: Using the stream programming model to describe *general purpose applications on GPUs* (GPGPU).
  - But some GPU-specific properties / limitations (not covered in this talk, see paper).

Background: Stream architectures

General restrictions:

- All memory write accesses take place at the *end* of a kernel

  → Limited number of outputs per kernel instance

  (since also the number of temporary registers per kernel instance is limited)

- Only (linear) stream writes, *no random access writes* !

universitätbonn TU Clausthal

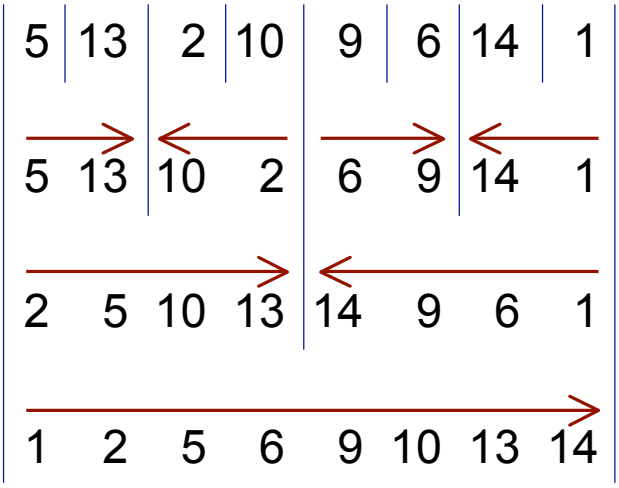
Recent work:
Sorting on stream architectures

- on Imagine:
  - Kapasi et al. 2000
- on GPUs:
  - Purcell et al. 2003
  - Kipfer et al. 2004 / 2005
  - Govindaraju et al. 2005

All based on non-optimal-time *sorting networks*,
most of them on Batcher´s bitonic sorting network.

universitätbonn TU Clausthal

Background:
Bitonic sorting



Standard merge sort scheme (log *n* merge steps),
but with alternating sorting directions

universitätbonn TU Clausthal

Background:
Bitonic merging

consists of log *n* steps (*stages*):

Split the (bitonic) input sequence into two equally sized bitonic sequences, such that all elements of the first sequence are not greater than any element of the second one.

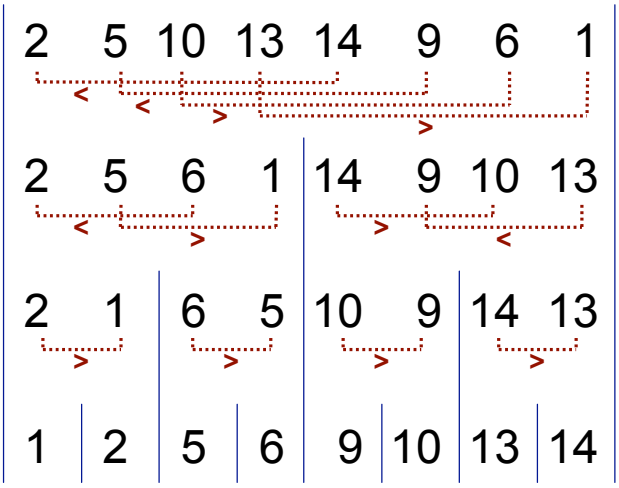
Bitonic sequence:

A sequence consisting of an increasing part followed by a descreasing part after rotation by an arbitrary number of elements.

e.g. 12343210, 32101234, 21012343

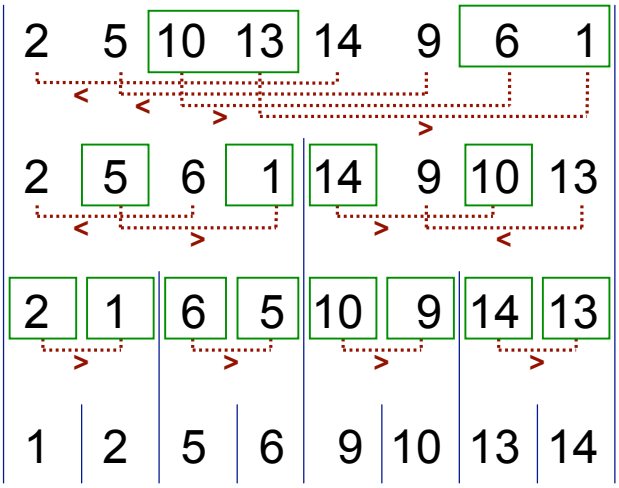
universitätbonn TU Clausthal

Background:
Bitonic merging



universitätbonn TU Clausthal

Background:
Bitonic merging



universitätbonn TU Clausthal

Background:
Adaptive bitonic merging

Each stage consists of log n steps (*phases*).

Idea:

- Find the devider of the partitions by binary search (instead of linear search).
- Use a binary search tree (*bitonic tree*).

  Perform exchanges by pointer swaps during the search.

universitätbonn TU Clausthal

Background:
A stage of the adaptive bitonic merge

Pseudo code:

Phase 0: Determine, which of the two cases applies:
(a) **root** value < **spare** value or
(b) **root** value > **spare** value
Only in case (b):
Exchange the values of root and spare.
Let **p** be the left and **q** the right son of root.

For i = 1, . . . , log n − 1:

Phase i: Test if: value of **p** > value of **q** (*)
If condition (*) is true:
Exchange the values of **p** and **q** as well as
in case (a) the left sons of **p** and **q**,
in case (b) the right sons of **p** and **q**.
Assign the left sons of **p**, **q** to **p**, **q** iff
case (a) applies and condition (*) is false or
case (b) applies and condition (*) is true;
otherwise assign the right sons of **p**, **q** to **p**, **q**.

universitätbonn TU Clausthal

Adaptive bitonic merging - First step towards a stream program

Basic ideas:

- Implement a single phase as kernel function.
- Store all temporary data to be transfered from one phase to another in a stream.

Phase 0: Determine, which of the two cases applies:
(a) **root** value < **spare** value or
(b) **root** value > **spare** value
Only in case (b):
Exchange the values of root and spare.
Let **p** be the left and **q** the right son of root.

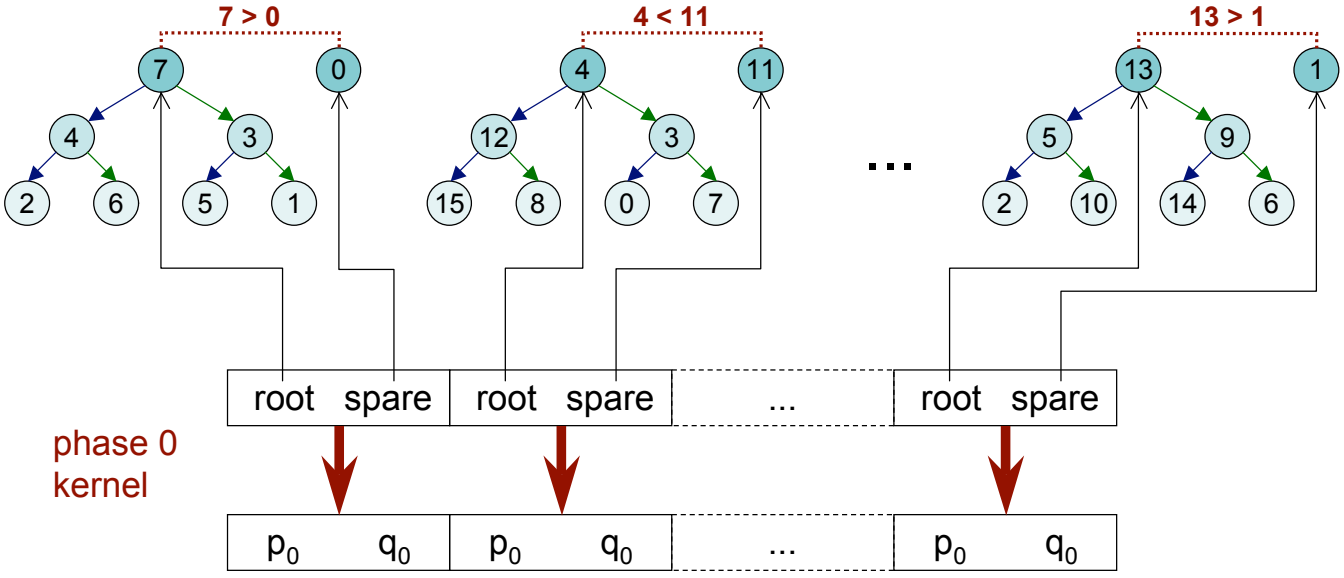
For i = 1, . . . , log n − 1:

Phase i: Test if: value of **p** > value of **q** (*)
If condition (*) is true:
Exchange the values of **p** and **q** as well as
in case (a) the left sons of **p** and **q**,
in case (b) the right sons of **p** and **q**.
Assign the left sons of **p**, **q** to **p**, **q** iff
case (a) applies and condition (*) is false or
case (b) applies and condition (*) is true;
otherwise assign the right sons of **p**, **q** to **p**, **q**.

universitätbonn TU Clausthal

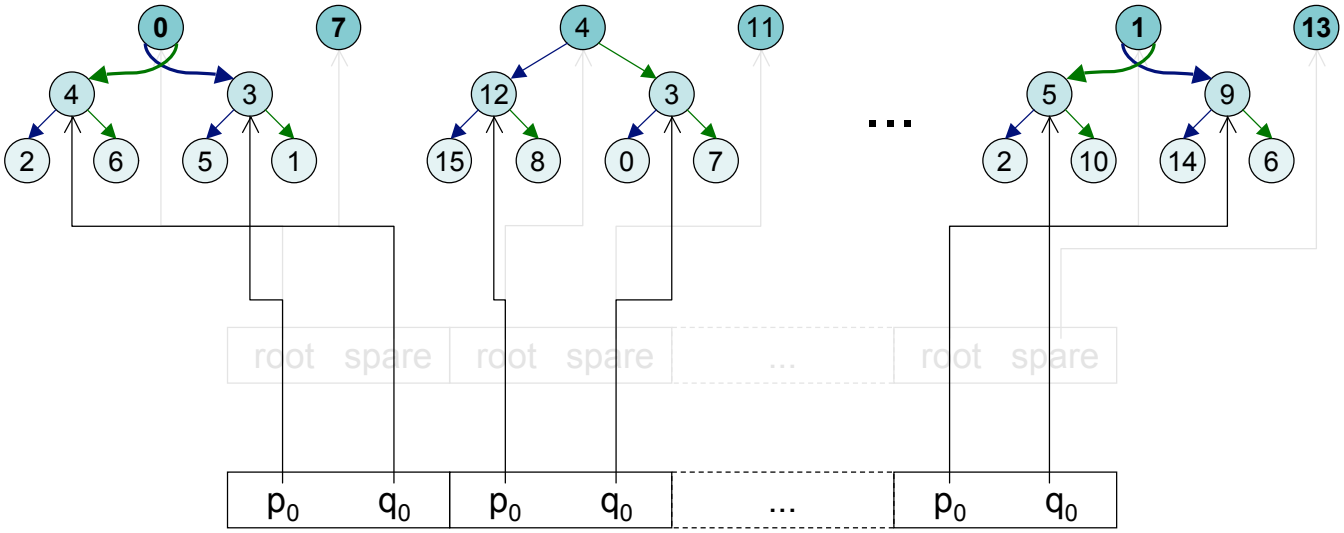
Adaptive bitonic merging - First step towards a stream program:

Execute phase 0 kernel...



Adaptive bitonic merging - First step towards a stream program:

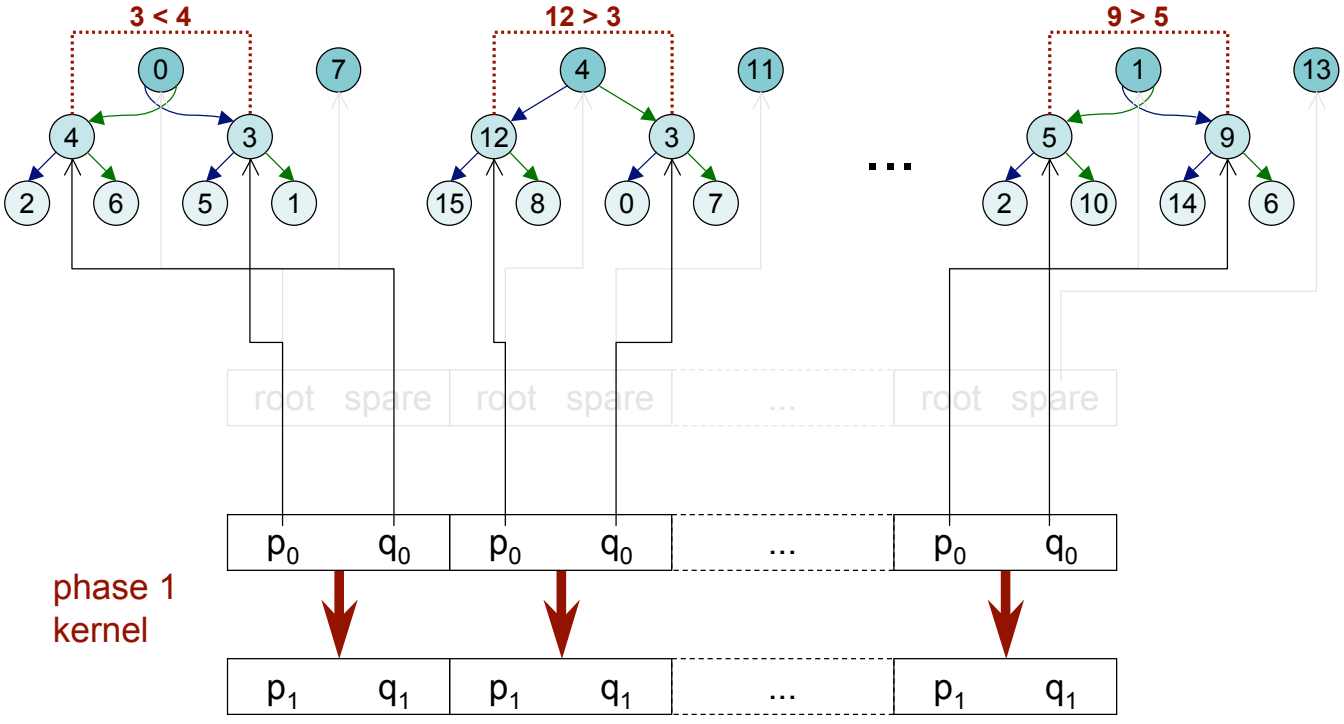
...result of phase 0 kernel...



universitätbonn TU Clausthal

Adaptive bitonic merging - First step towards a stream program:

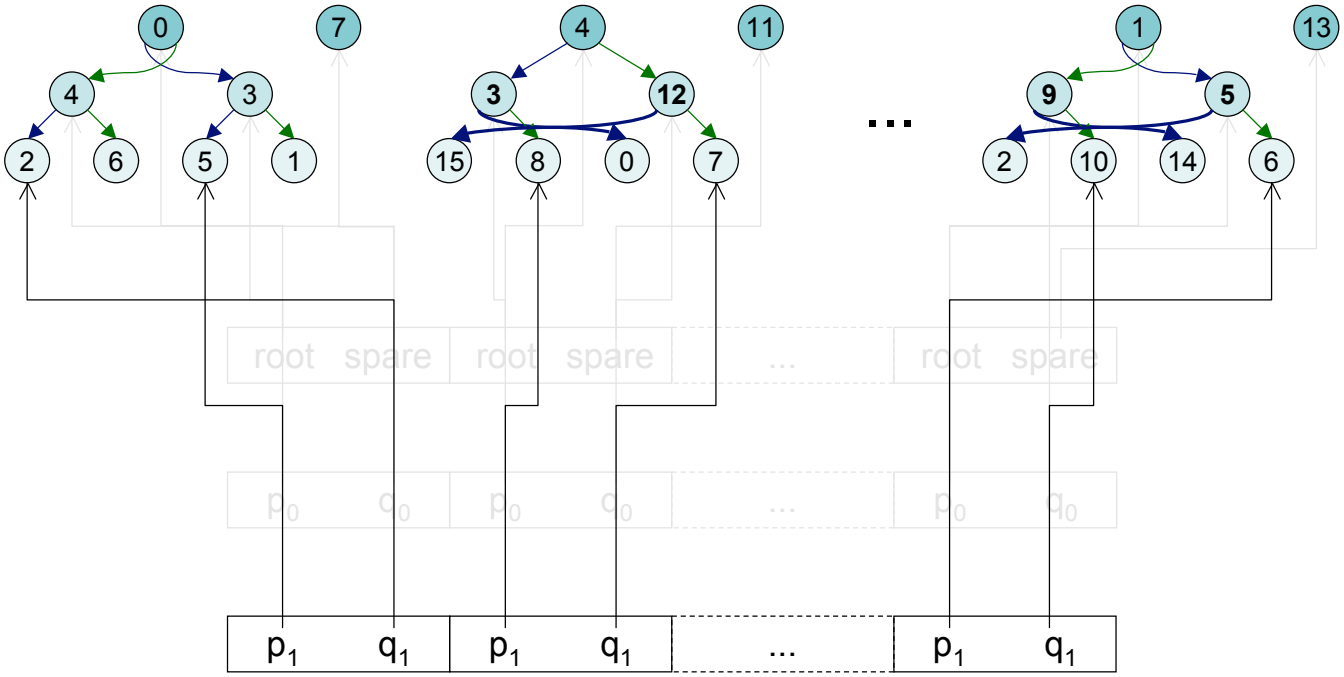
...execute phase 1 kernel...



universitätbonn TU Clausthal

Adaptive bitonic merging - First step towards a stream program:

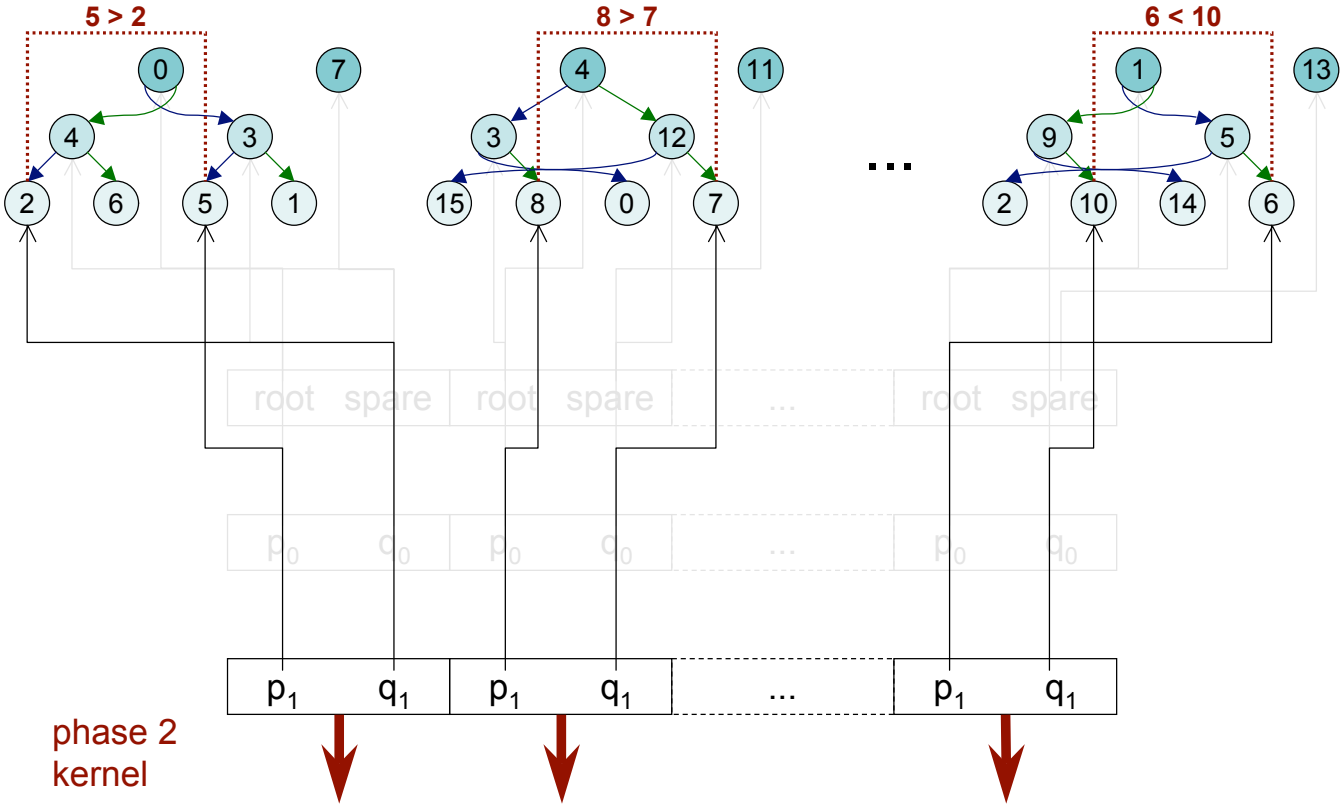
...result of phase 1 kernel...



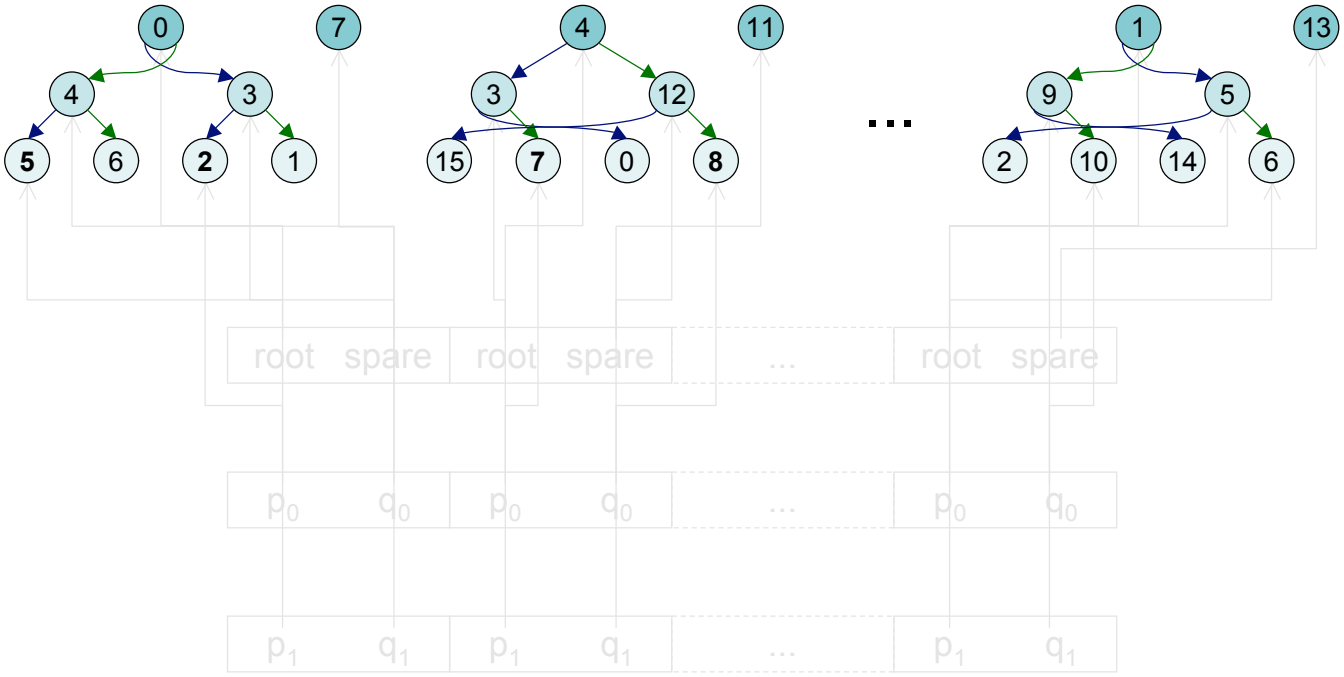
universitätbonn TU Clausthal

Adaptive bitonic merging - First step towards a stream program:

...execute phase 2 kernel...



...first merge stage done



Recall the stream programming restrictions:

- All memory write accesses take place at the *end* of a kernel

  → Limited number of outputs per kernel instance

✓

- Only (linear) stream writes, *no random access writes* !

✗

Adaptive bitonic merging - Actual stream program:
Stream writes instead of random access writes

- Can we output the modified nodes linearly to a stream (i.e. in the order they are visited)?

  Recall:
  - The order in which nodes are visited is data dependent.
  - The number of nodes visited is not data dependent.

- We can, as long as we keep child pointers consistent.
  - Because we operate on a fully linked data structure: the bitonic *tree*.
  - i.e. we can change physical memory location of nodes during the algorithm if we update the corresponding child pointers.

universitätbonn TU Clausthal

Adaptive bitonic merging - Actual stream program:
Stream writes instead of random access writes

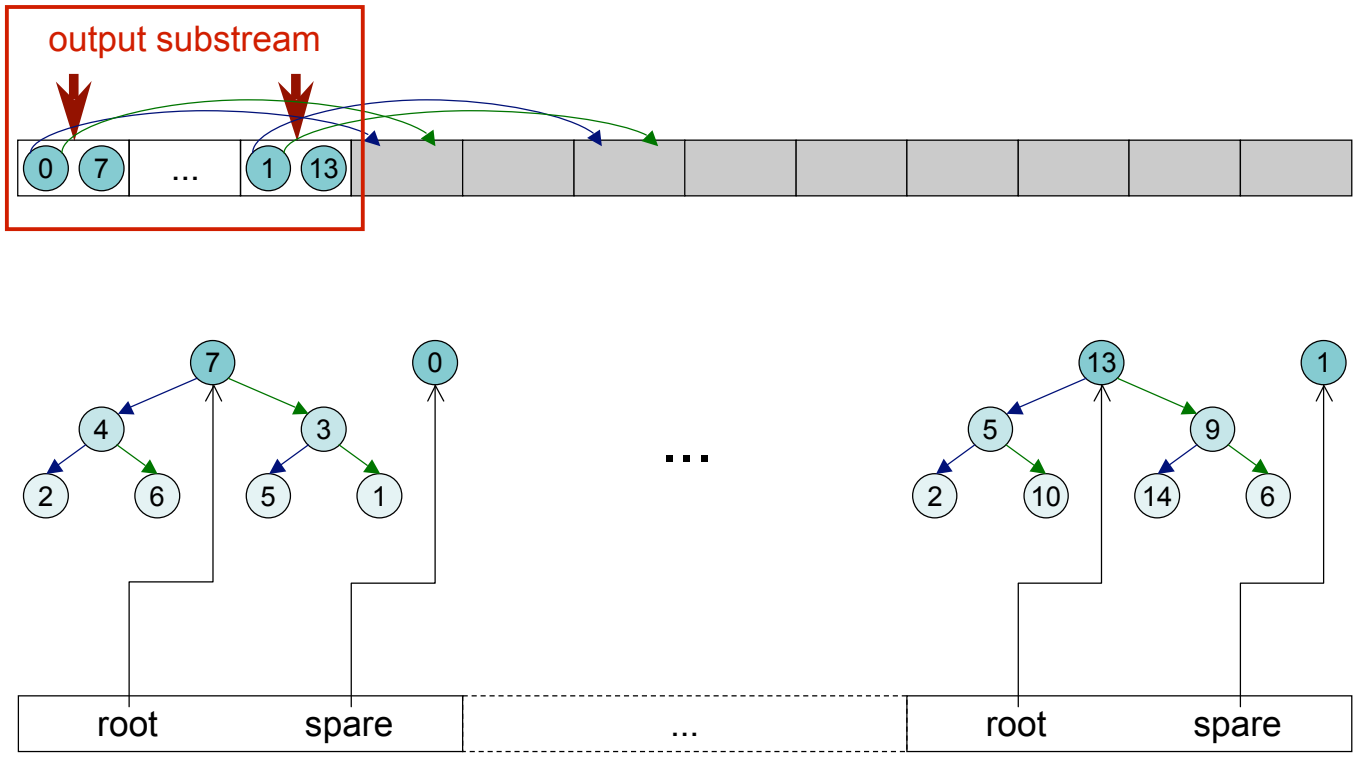
- But how to update the child pointers *without* random access writes?
  - At the end of each phase, we know which child nodes will be visited in the next phase.
  - We also know to which memory location the modified child nodes will be stored in the next phase (because of the fixed memory layout).

- Therefore, we can update the child pointers *in advance* (when creating the parent node).

universitätbonn TU Clausthal

Adaptive bitonic merging - Actual stream program:

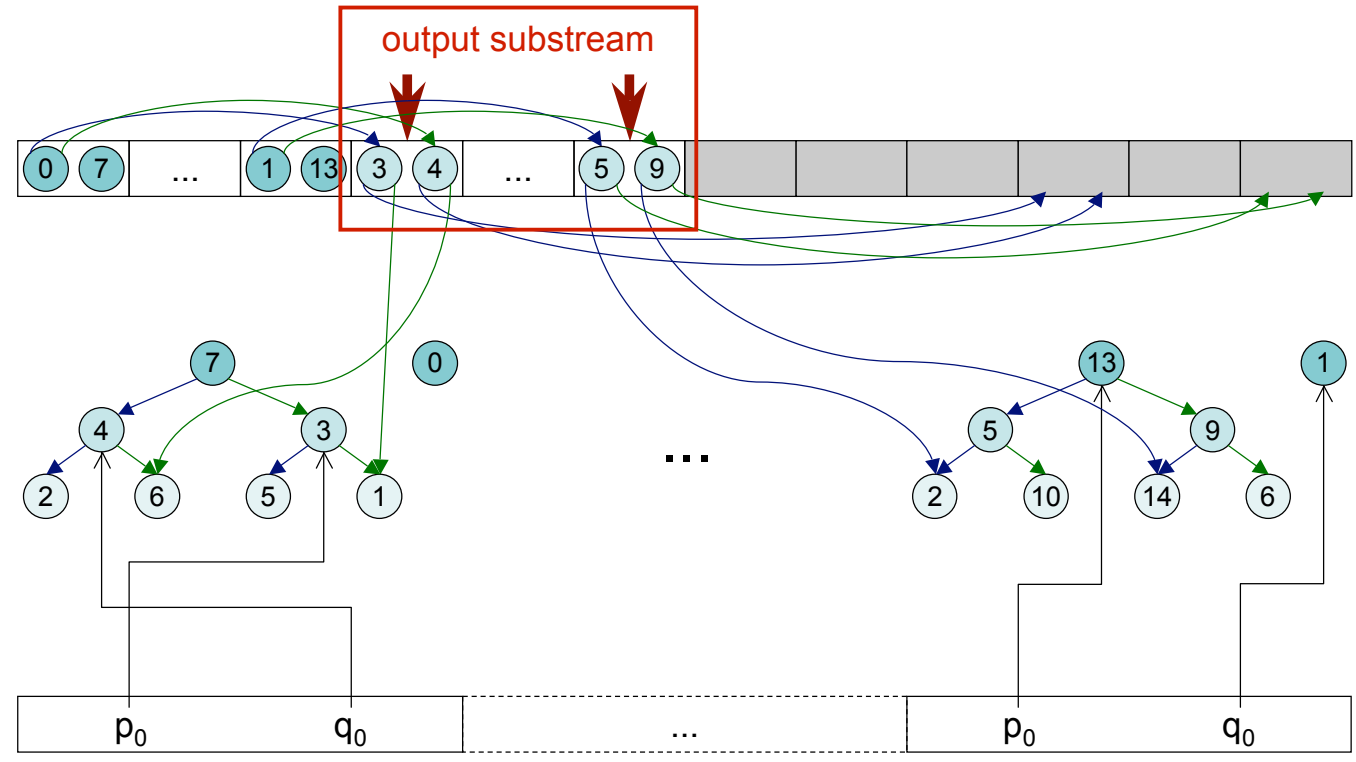
Execute phase 0 kernel...



universitätbonn TU Clausthal

Adaptive bitonic merging - Actual stream program:

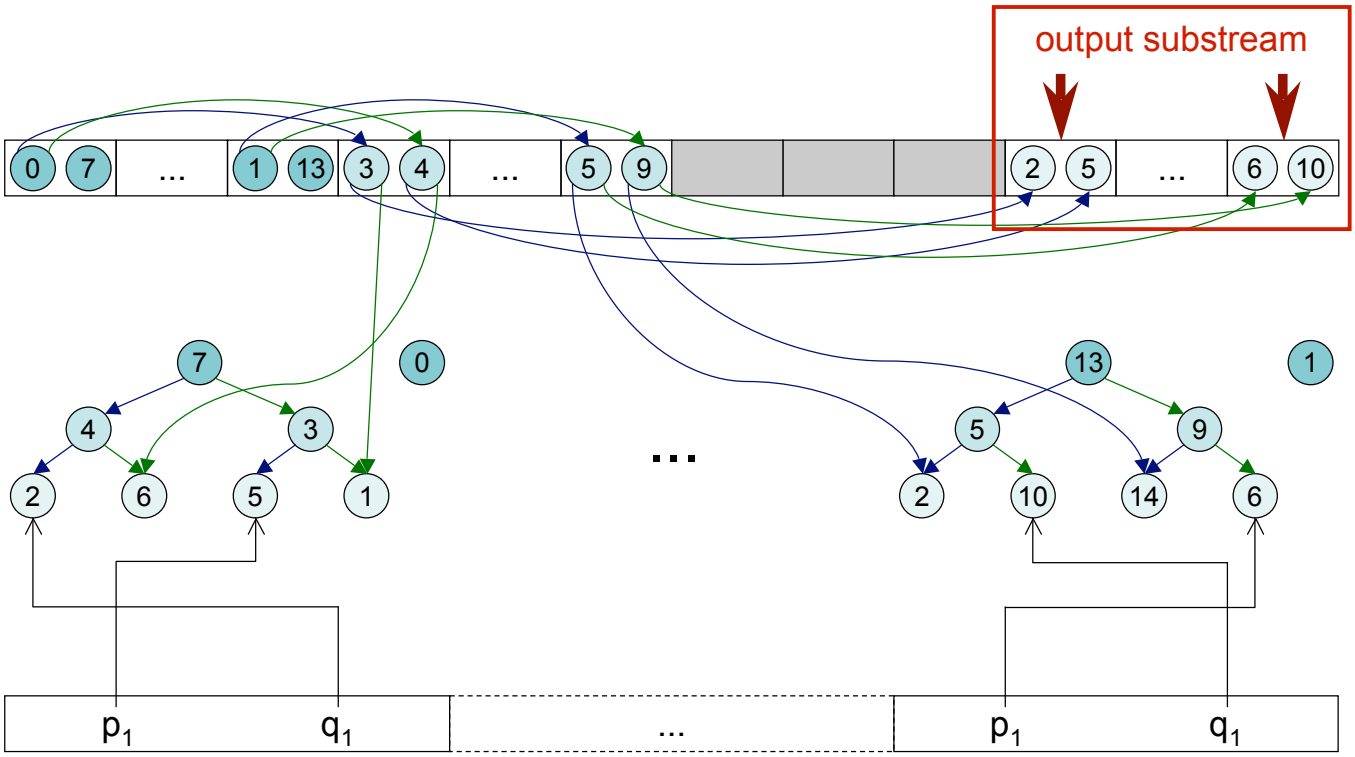
...execute phase 1 kernel...



universitätbonn TU Clausthal

Adaptive bitonic merging - Actual stream program:

...execute phase 2 kernel...



universitätbonn TU Clausthal

Adaptive bitonic merging:

Overlapped execution of phases

Additional improvement to reduce the number of *stream operations* (kernel function calls) from $O(\log^3 n)$ to $O(\log^2 n)$ (similar to Bilardi et al.).

See paper.

universitätbonn TU Clausthal

Implementation issues

- Optimization:

  The first few merge steps as well as the last few stages of each merge can be replaced by specially optimized merge implementations for small sequences

  (i.e. sequences up to 16 elements in our implementation).

- GPU-specific implementation issues:
  - ensure distinctness of input and output streams (currently requires additional copying of substreams)
  - map all pointers / indexes to the 2D address space of GPU memory; alternatives:
    1) simple row-wise mapping
    2) GPU cache optimized 1D-to-2D address mapping

universitätbonn TU Clausthal

Results

CPU: AMD Athlon-XP 3000+ GPU: NVIDIA GeForce 6800 Ultra

n	CPU std::sort()	Govindaraju et al. 2005	GPU-ABiSort[1]	GPU-ABiSort[2]
32768	12 – 16 ms	13 ms	11 ms	8 ms
65536	27 – 35 ms	29 ms	21 ms	16 ms
131072	62 – 77 ms	63 ms	45 ms	31 ms
262144	126 – 160 ms	139 ms	95 ms	64 ms
524288	270 – 342 ms	302 ms	208 ms	133 ms
1048576	530 – 716 ms	658 ms	479 ms	279 ms

CPU: AMD Athlon-64 4200+ GPU: NVIDIA GeForce 7800 GTX

n	CPU std::sort()	Govindaraju et al. 2005	GPU-ABiSort[1]	GPU-ABiSort[2]
32768	9 – 11 ms	4 ms	6 ms	5 ms
65536	19 – 24 ms	8 ms	9 ms	8 ms
131072	46 – 52 ms	18 ms	18 ms	16 ms
262144	98 – 109 ms	38 ms	37 ms	31 ms
524288	203 – 226 ms	80 ms	76 ms	65 ms
1048576	418 – 477 ms	173 ms	165 ms	135 ms

[1]) simple mapping to 2D address space

[2]) cache-optimized mapping to 2D address space

universitätbonn TU Clausthal

Thank you.

universitätbonn TU Clausthal