PMPP 2015/16

Parallel Sorting

(Preliminary) Course Schedule

you are here → 9.11.2015

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Lecture Parallel Basics, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A), S3|19, Room 2.8
2.11.2015	Intro Final Proj., Ex. 1 due, Ex. 2 assigned, Lecture PRAM
3.11.2015	Lecture PRAM (2)
9.11.2015	Final Projects assigned, L. Parallel Sort., Exercise 2 due
10.11.2015	Lecture
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)

Final Projects

- first presentation on 23.11.2014 and 24.11.2014
  - present the topic and the approach you plan to use to the class
  - overview over the problem
  - proposed solution using CUDA
  - everybody in class should understand the problem and the proposed solution!
  - submit slides ahead of time for faster change between presenters
- meet with your advisor early enough
- total time 20 minutes per topic including questions (for all teams together)
- everybody should give a portion of the presentation
- mandatory (talk to us if this is a problem)

Final Projects Assignment (Moodle)

Member1	Member2	Topic
Szymon Michalski	Jacopo Abramo	Nonlinear Magnetoquasistatic Spectral-Element Solver
Ashkan Ashooripour Moghadam	Maxime Grauer	Nonlinear Magnetoquasistatic Spectral-Element Solver
Yannic Fischler	Felix Schuwirth	Nonlinear Magnetoquasistatic Spectral-Element Solver
David Winter	Sebastian Focke	Fixed-Rate Compressed Floating-Point Arrays
Puneet Arora	Prabhjot Singh	Fixed-Rate Compressed Floating-Point Arrays
Patrick Adler	Tobias Erbshäußer	Financial Time Series
Florian Saumweber	ASIT KUMAR DHAL	Financial Time Series
Florian Lang	Maximilian Albert Weigel	Relativistic Particle-Particle Interaction in Linear Accelerators
Mallikarjun Nuti	Gaurav Singh	Relativistic Particle-Particle Interaction in Linear Accelerators
Sreeram Sadasivam	Aakash Sharma	Financial Time Series
Daniel Elias Roth	Tobias Averhage	Relativistic Particle-Particle Interaction in Linear Accelerators
Fabio Arnold	Ravi Sarangdhar	Real-time Meshless Physics
Matthias Hofmann	Dominique Patrick Metz	Fixed-Rate Compressed Floating-Point Arrays
Florian Zouhar	Raffael Haberland	Real-time Meshless Physics
Shrikanth Malavalli Divakar	Arun Kumar Naranahalli Anjanappa	Real-time Meshless Physics
Lukas Graner	David Bug	Data Term Computation for Texture Optimization
Sanjaya Subedi	Kushal Gautam	Sequence Clustering for Bio-Informatics
Sachin Kumar	Marc-Pascal Peter Clement	Sequence Clustering for Bio-Informatics
Lars Fritsche	Daniel Jente	Adaptive Manifolds for Real-Time High-Dimensional Filtering
Daniel Kauth	Imaginary Friend	Adaptive Manifolds for Real-Time High-Dimensional Filtering
Julián-David Torregrosa	Andrés Felipe Mesa-Gutiérrez	Adaptive Manifolds for Real-Time High-Dimensional Filtering
Viswanath Vadhri	Sumit Sati	Data Term Computation for Texture Optimization
Patrick Seemann	Nils Möhrle	Data Term Computation for Texture Optimization
Sudeep Duggal	Muhammad Rameez	Sequence Clustering for Bio-Informatics
Madhukar Hassan Chinnappa	Manikandan Ravichandran	Relativistic Particle-Particle Interaction in Linear Accelerators

Written Exam

- Date for the exam will be 02.03.2016, starting at 2pm
- Rooms: S101/A1 and S101/A01
- The exam will be in english only

Parallel Sorting Algorithms

- parallel sorting (fixed key size)
  - bucket sort
  - radix sort

  ➔ both require prefix sum to compute intermediate information

  - bitonic sort

  ➔ approach well suited for hardware implementations

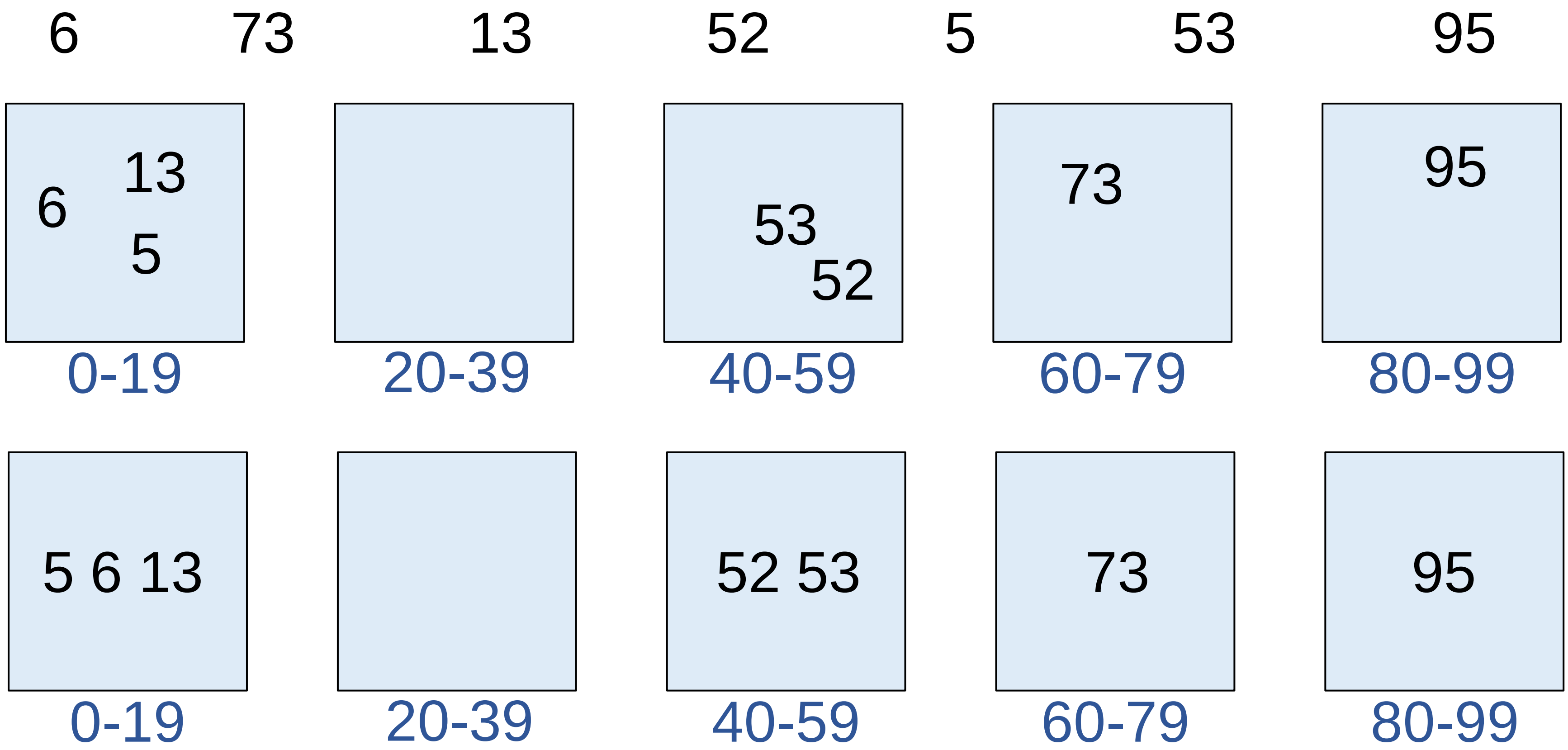
  - sorting in practice

Bucket Sort

- set up an array of initially empty „buckets“
- go over the original array, putting each object in its bucket → scatter
- sort each non-empty bucket
- visit the buckets in order and put all elements back into the original array → gather

slides partially by H. Lensch/R. Strzodka

Bucket Sort Example



5 6 13 52 53 73 95

slides partially by H. Lensch/R. Strzodka

Parallel Bucket Sort

- compute histogram
  - e.g. using atomics (or following the example in the SDK)
  - `idx = atomicsInc()`

- compute Prefix Sum of histogram bins
  - calculating the starting index of each bin –> `bin_start`

- write each element in parallel to (`bin_start + idx`)

- now sort within bucket
  - each bucket in parallel

slides partially by H. Lensch/R. Strzodka

Radix Sort

- iterated bucket sort on individual digits
- requires stable sorting

Radix Sort

- review of serial version
  - sort from the least significant bit to the most significant bit using counting sort
- initial situation
- after sorting on 2nd digit
- after sorting on 1st digit

89	28	81	69	14	31	29	18	39	17
81	31	14	17	28	18	89	69	29	39
14	17	18	28	29	31	39	69	81	89

- complexity: O(n)
  - fast
  - simple to code
  - requires stable sorting

slides partially by H. Lensch/R. Strzodka

Parallel Implementation

- main problem: keep sorting stable
- compute radix histogram in each block
- use PrefixSum to calculate offset for each bin and scatter
- iterate

slides partially by H. Lensch/R. Strzodka

Parallel Implementation

```
for (uint shift = 0; shift < bits; shift += RADIX) {
    // Perform one round of radix sorting

    // Generate per radix group sums radix
    // counts across a radix group
    RadixSum<<<...>>>(pData0, elements, shift);

    // Prefix sum in radix groups, and then
    // between groups throughout a block
    RadixPrefixSum<<<...>>>();

    // Sum the block offsets and then shuffle
    // data into bins
    RadixAddOffsetsAndShuffle<<<...>>>
        (pData0, pData1, elements, shift);

    // Exchange data pointers
    KeyValuePair* pTemp = pData0;
    pData0 = pData1;
    pData1 = pTemp;
}
```

[code from SDK]

slides partially by H. Lensch/R. Strzodka

Review of Parallel Radix Sort

- outer loop is independent of the problem size
- parallel complexity: O(1)
  - given a sufficient number of processors
- problem:
  - current approaches write the entire array multiple times

slides partially by H. Lensch/R. Strzodka

Definition: Sorting Network

- A sorting network is an algorithm that sorts a fixed number of values using a fixed sequence of comparisons.
  - They can be thought of as networks of wires and comparator modules. Values (of any ordered type) flow across the wires.
  - The comparators each connect two wires, compare the values coming in on the wires, and sort them by outputting the smaller value to one wire, and the larger to the other.

- Sorting networks differ from general comparison sorts in that they are not capable of handling arbitrarily large inputs, and in that their sequence of comparisons is set in advance, regardless of the outcome of previous comparisons.
  - This independence of comparison sequences is useful for parallel execution and for implementation in hardware.

0-1 Lemma

- Theorem: (0-1-principle)
  - If a sorting network sorts every sequence of 0's and 1's, then it sorts every arbitrary sequence of values.
  - The proof of the 0-1-principle is not very difficult. However, it is quite helpful to have some definitions and lemmas ready.

- Note:
  - This is not applicable to bucket/radix sort style of algorithms.
  - It works for all algorithms that swap entries based on comparisons.

Proof from: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm

0-1 Lemma Preliminaries

- Definition
  - Let A and B be ordered sets. A mapping $f: A \rightarrow B$ is called monotonic if for all $a_1, a_2 \in A$
    - $a_1 \leq a_2 \Longrightarrow f(a_1) \leq f(a_2)$
- Lemma
  - Let $f: A \rightarrow B$ be a monotonic mapping. Then the following holds for all $a_1, a_2 \in A$:
    - $f(\min(a_1, a_2)) = \min\big(f(a_1), f(a_2)\big)$
- Proof
  - Let $a_1 \leq a_2$ and thus $f(a_1) \leq f(a_2)$. Then
    - $\min(a_1, a_2) = a_1$ and $\min\big(f(a_1), f(a_2)\big) = f(a_1)$
  - This implies
    - $f(\min(a_1, a_2)) = f(a_1) = \min\big(f(a_1), f(a_2)\big)$
  - Similarly, if $a_2 \leq a_1$ and therefore $f(a_2) \leq f(a_1)$, we have
    - $f(\min(a_1, a_2)) = f(a_2) = \min\big(f(a_1), f(a_2)\big)$
- An analogous property holds for the max-function.

0-1 Lemma Preliminaries

- Definition
  - Let $f: A \rightarrow B$ be a mapping. The extension of f to finite sequences $a = a_0, \dots, a_{n-1}, a_i \in A$ is defined as follows:
    - $f(a_0, \dots, a_{n-1}) = f(a_0), \dots, f(a_{n-1})$, i.e.
    - $f(a)_i = f(a_i)$
- Lemma
  - Let f be a monotonic mapping and N a comparator network. Then N and f commutate, i.e. for every finite sequence $a = a_0, \dots, a_{n-1}$ the following holds:
    - $N\big(f(a)\big) = f\big(N(a)\big)$
  - In other words: a monotonic mapping f can be applied to the input sequence of comparator network N or to the output sequence, the result is the same.

0-1 Lemma Preliminaries

TECHNISCHE UNIVERSITÄT DARMSTADT

- Proof
  - For a single comparator $[i{:}\,j]$ the following holds (see definition of comparator):
    - $[i{:}\,j]\big(f(a)\big)_i = [i{:}\,j]\big(f(a_0), \dots, f(a_{n-1})\big)_i = \min\Big(f(a_i), f\big(a_j\big)\Big) = f\big(\min\big(a_i, a_j\big)\big) = f([i{:}\,j](a)_i) = f\big([i{:}\,j](a)\big)_i$
  - This means that the i^{th} element is the same regardless of the order of application of f and $[i{:}\,j]$.
  - The same can be shown for the j^{th} element and for all other elements. Therefore
    - $[i{:}\,j]\big(f(a)\big) = f\big([i{:}\,j](a)\big)$
  - For an arbitrary comparator network N (which is a composition of comparators) and a monotonic mapping f we have therefore
    - $N\big(f(a)\big) = f\big(N(a)\big)$

0-1 Lemma Proof

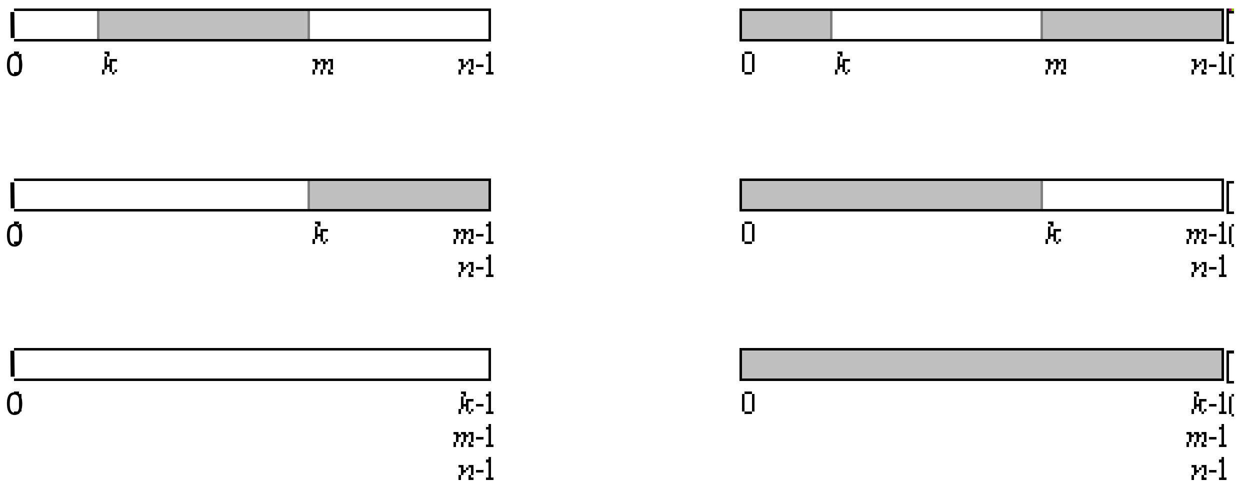
- Theorem: (0-1-principle)
  - Let N be a comparator network. If every 0-1-sequence is sorted by N, then every arbitrary sequence is sorted by N.
- Proof
  - Suppose a with $a_i \in A$ is an arbitrary sequence which is not sorted by N. This means $N(a) = b$ is unsorted, i.e. there is a position k such that $b_k > b_{k+1}$.
  - Now define a mapping $f: A \rightarrow \{0, 1\}$ as follows.
  - For all c element A let
    - $f(c) = \begin{cases} 0 & \text{if } c < b_k \\ 1 & \text{if } c \geq b_k \end{cases}$

0-1 Lemma Proof

- Proof
  - Obviously, f is monotonic. Moreover we have:
    - $f(b_k) = 1$ and $f(b_{k+1}) = 0$
  - i.e. $f(b) = f\big(N(a)\big)$ is unsorted.
  - This means that $N\big(f(a)\big)$ is unsorted or, in other words, that the 0-1-sequence $f(a)$ is not sorted by the comparator network N.
  - We have shown that, if there is an arbitrary sequence a that is not sorted by N, then there is a 0-1-sequence $f(a)$ that is not sorted by N.
  - Equivalently, if there is no 0-1-sequence that is not sorted by N, then there can be no sequence a whatsoever that is not sorted by N.
  - Equivalently again, if all 0-1-sequences are sorted by N, then all arbitrary sequences are sorted by N. ■

Bitonic Sequence

- binary sequence with at most two transitions at k and m



- sequence A = $a_0, a_1, \ldots, a_{n-1}$ is bitonic if
  - there is an index i, $0 < i < n$, s.t.
    $a_0 .. a_i$ is increasing and $a_i .. a_{n-1}$ is decreasing
  - or there is a cyclic shift of A for which 1 holds.

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort

- first step:
  build an efficient network to sort a bitonic sequence

- general idea:
  subdivide the sequence into two halfs
  - one half is bitonic and already sorted (clean)
  - other half is only bitonic

- reminder:
  we are handling binary sequences only
  (elements are 0 or 1)
  - it can be shown that a sorting network that is correct for all binary sequences works also for general input sequences

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Split

- a bitonic split divides a bitonic sequence in two:

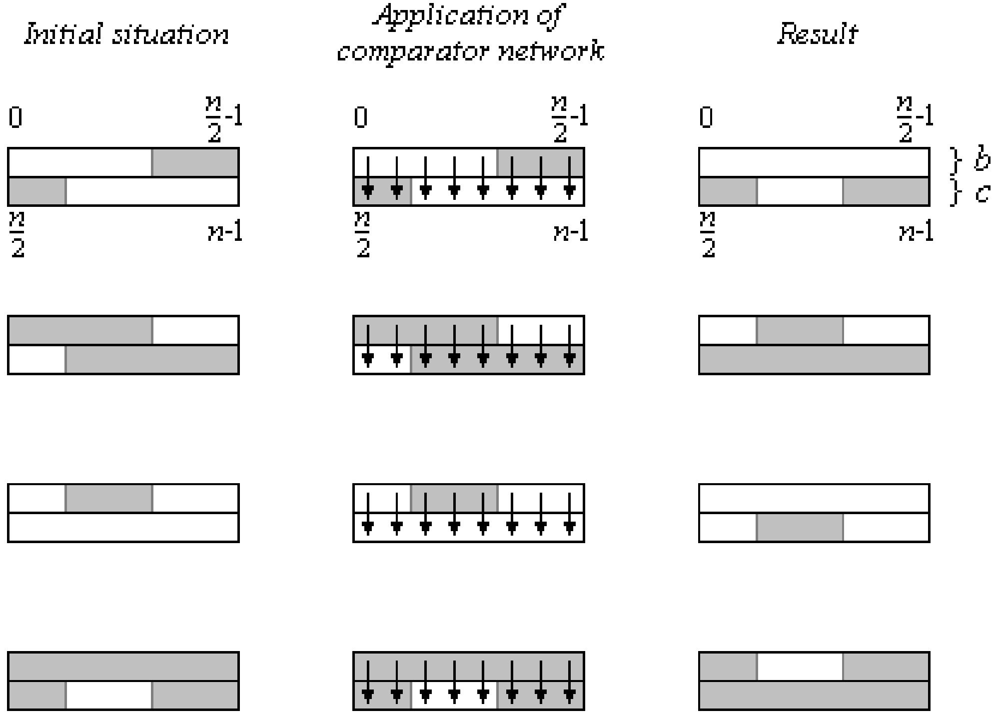
$$BitSplit(BS) = \begin{cases} S_1 = \left(\min(bs_0, bs_{\frac{n}{2}}), \ldots, \min(bs_{\frac{n}{2}-1}, bs_{n-1})\right) \\ S_2 = \left(\max(bs_0, bs_{\frac{n}{2}}), \ldots, \max(bs_{\frac{n}{2}-1}, bs_{n-1})\right) \end{cases}$$

- theorem:
  - S1 and S2 are both bitonic
  - S1(i) <= S2(j) for all i,j in 0,..,n/2

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Split – Comparison Network

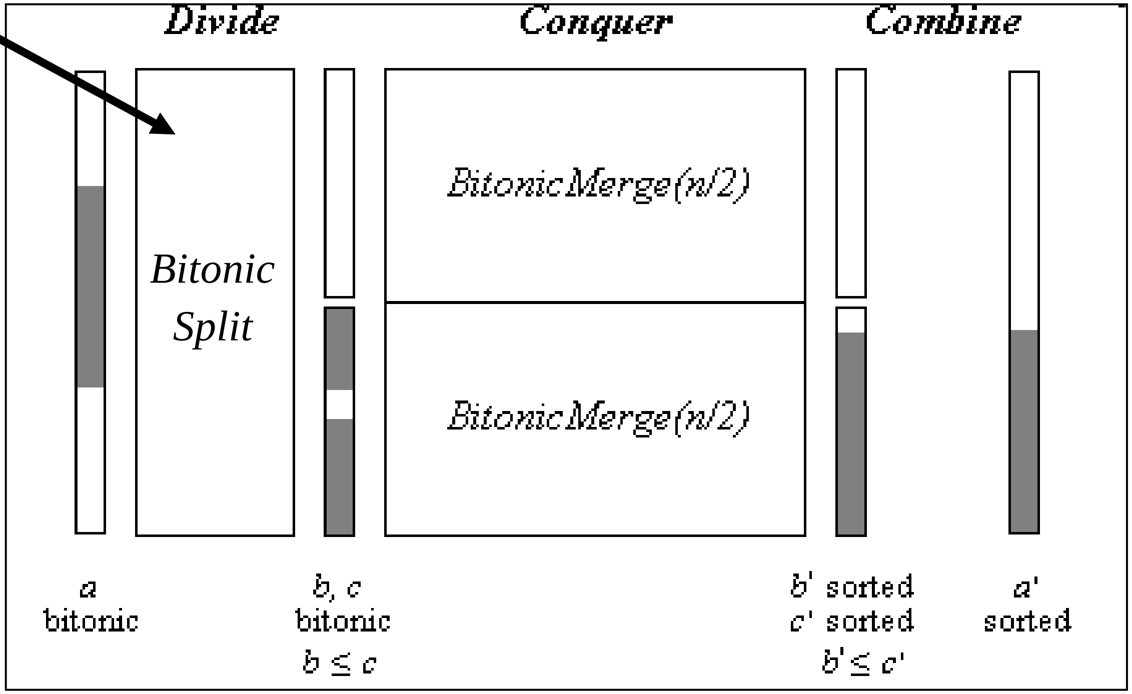
- given two bitonic sets bn, cn perform an element wise comparison storing the smaller (larger) value in b (c).
- output is guaranteed to be bitonic again.



slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Merge

- given a bitonic vector, it will be sorted by applying a comparison network (divide) to its to halves
- then recurse

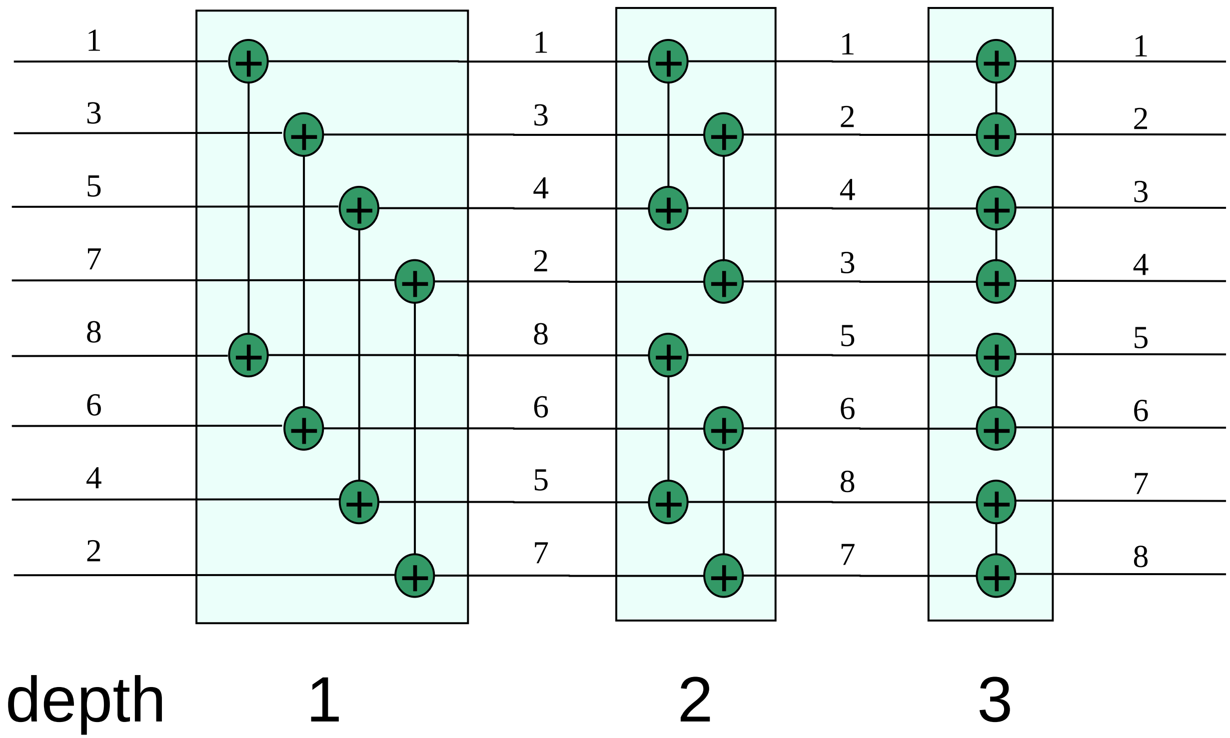


BitonicMerge(n)

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Merge

- given: a bitonic sequence BS of size n = 2m
- sort BS using m (parallel) Bitonic Splits stages



slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort

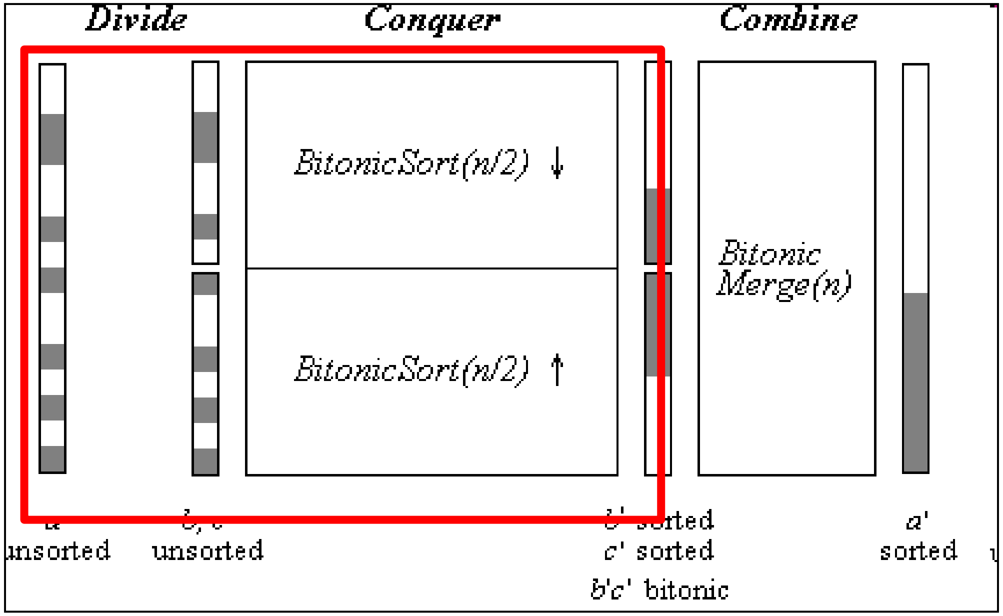
- first step:
  build an efficient network to sort a bitonic sequence ✓

- second step:
  build an efficient network to sort a general (binary) sequence)

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort

- split input vector into two halves
- sort halves ascending/descending and combine them
- sort the resulting bitonic vector using bitonic merge

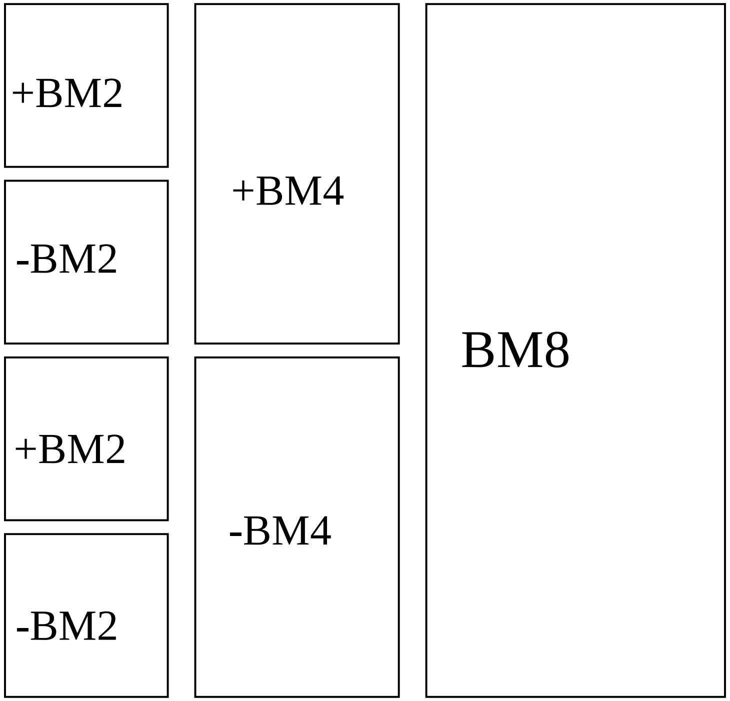


BitonicSort(n)

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort

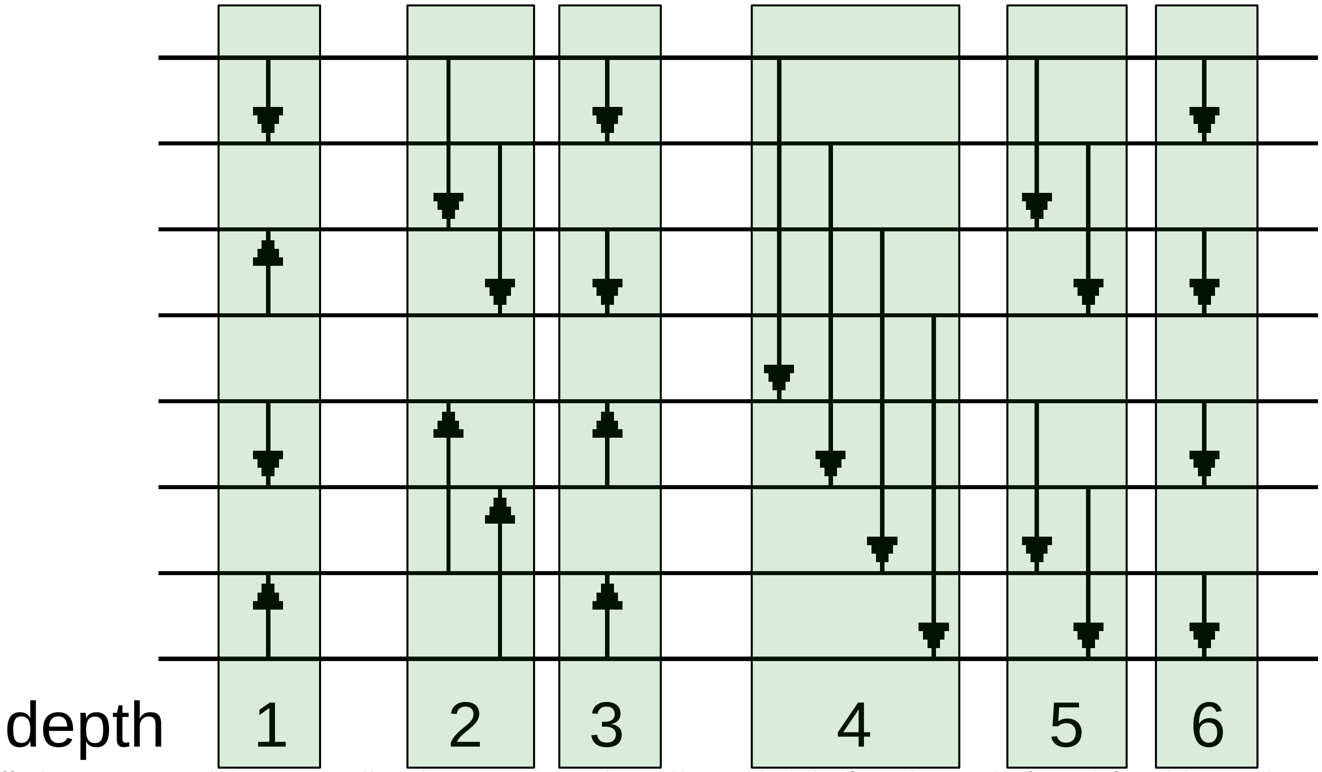
- bitonic sort = log(n) bitonic merge stages
  - note that any binary vector with length 2 is bitonic
- same number of comparison operations in each iteration



slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort

- bitonic sort = log(n) bitonic merge stages
- same number of comparison operations in each iteration



slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Bitonic Sort – Summary

- complexity $O(n \log(n)^2)$
  - there might be a lower bound than this but only in theory
  - the design of optimal sorting networks is co-NP complete
- fixed comparison networks
  - they are all operating on the list in parallel
- simple to implement

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

Code from SDK

```
// Copy input to shared mem.
shared[tid] = values[tid];
__syncthreads();

// Parallel bitonic sort.
for (int k = 2; k <= NUM; k *= 2) {
    // Bitonic merge:
    for (int j = k / 2; j>0; j /= 2) {
        int ixj = tid ^ j;  // XOR
        if (ixj > tid){
            if ((tid & k) == 0) { // ascending - descending
                if (shared[tid] > shared[ixj])
                    swap(shared[tid], shared[ixj]);
            } else {
                if (shared[tid] < shared[ixj])
                    swap(shared[tid], shared[ixj]);
            }
        }
        __syncthreads();
    }
}
values[tid] = shared[tid]; // Write result.
```

slides partially by H. Lensch/R. Strzodka, images from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm

0.02$ on Sorting in Practice

- There are several ways of sorting in CUDA in practice
- Option 1:
  - Use Thrust
- Option 2:
  - Someone else already published a sorting algorithm that fits your needs
- Option 3:
  - Write your own kernel
  - Avoid this one at all costs

Sorting in Thrust

3.5. Sorting

Thrust offers several functions to sort data or rearrange data according to a given criterion. The `thrust::sort` and `thrust::stable_sort` functions are direct analogs of sort and stable_sort in the STL.

```
#include <thrust/sort.h>

...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};

thrust::sort(A, A + N);

// A is now {1, 2, 4, 5, 7, 8}
```

Sorting in Thrust

In addition, Thrust provides `thrust::sort_by_key` and `thrust::stable_sort_by_key`, which sort key-value pairs stored in separate places.

```
#include <thrust/sort.h>

...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);

// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Sorting in Thrust

Like their STL brethren, the sorting functions also accept user-defined comparison operators:

```
#include <thrust/sort.h>
#include <thrust/functional.h>

...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};

thrust::stable_sort(A, A + N, thrust::greater<int>());

// A is now {8, 7, 5, 4, 2, 1}
```

Literature on PRAM Algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

- Practical PRAM Programming
  Jörg Keller, Christoph W. Kessler, Jesper Larsson Träff
  Wiley Series on Parallel and Distributed Computing, 2001
  - unfortunately out of print

- Synthesis of Parallel Algorithms, J. H. Reif
  Morgan Kaufmann Publishers, 1993

- An Introduction to Parallel Algorithms, J. Jàjà
  Addison-Wesley, 1992

Recommended Reading

- presentation on parallel reduction using CUDA by Mark Harris
  - step-by-step discussion of optimizations to create a high-performance implementation of reduction
  - available in the doc directory of the reduction SDK example
  - will be added in Moodle as well