GPU Sample Sort

Nikolaj Leischner, *Vitaly Osipov*, Peter Sanders

Institut für Theoretische Informatik - Algorithmik II

Overview

- Introduction
- Tesla architecture
- Computing Unified Device Architecture Model
- Performance Guidelines
- Sample Sort Algorithm Overview
- High Level GPU Algorithm Design
- Flavor of Implementation Details
- Experimental Evaluation
- Future Trends

Introduction

multi-way sorting algorithms

- Sorting is important
- Divide-and-Conquer approaches:
  - recursively split the input in tiles until the tile size is M (e.g cache size)
  - sort each tile independently
  - combine intermidiate results
- Two-way approaches:
  - two-way distribution - quicksort $\rightsquigarrow \log_2(n/M)$ scans to partition the input
  - two-way merge sort $\rightsquigarrow \log_2(n/M)$ scans to combine intermidiate results
- Multi-way approaches:
  - k-way distribution - sample sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to partition
  - k-way merge sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to combine
- Multiway approaches are benifitial when the memory bandwidth is an issue!

Introduction

multi-way sorting algorithms

- Sorting is important
- Divide-and-Conquer approaches:
  - recursively split the input in tiles until the tile size is M (e.g cache size)
  - sort each tile independently
  - combine intermidiate results
- Two-way approaches:
  - two-way distribution - quicksort $\rightsquigarrow \log_2 (n/M)$ scans to partition the input
  - two-way merge sort $\rightsquigarrow \log_2 (n/M)$ scans to combine intermidiate results
- Multi-way approaches:
  - k-way distribution - sample sort $\rightsquigarrow$ only $\log_k (n/M)$ scans to partition
  - k-way merge sort $\rightsquigarrow$ only $\log_k (n/M)$ scans to combine
- Multiway approaches are benifitial when the memory bandwidth is an issue!

Introduction

multi-way sorting algorithms

- Sorting is important
- Divide-and-Conquer approaches:
  - recursively split the input in tiles until the tile size is M (e.g cache size)
  - sort each tile independently
  - combine intermidiate results
- Two-way approaches:
  - two-way distribution - quicksort $\rightsquigarrow \log_2(n/M)$ scans to partition the input
  - two-way merge sort $\rightsquigarrow \log_2(n/M)$ scans to combine intermidiate results
- Multi-way approaches:
  - k-way distribution - sample sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to partition
  - k-way merge sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to combine
- Multiway approaches are benifitial when the memory bandwidth is an issue!

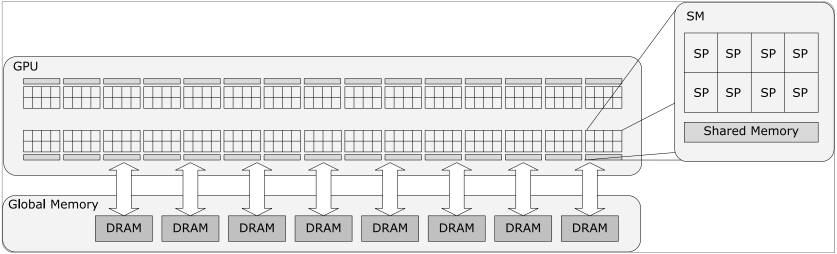
Introduction

multi-way sorting algorithms

- Sorting is important
- Divide-and-Conquer approaches:
  - recursively split the input in tiles until the tile size is M (e.g cache size)
  - sort each tile independently
  - combine intermidiate results
- Two-way approaches:
  - two-way distribution - quicksort $\rightsquigarrow \log_2(n/M)$ scans to partition the input
  - two-way merge sort $\rightsquigarrow \log_2(n/M)$ scans to combine intermidiate results
- Multi-way approaches:
  - k-way distribution - sample sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to partition
  - k-way merge sort $\rightsquigarrow$ only $\log_k(n/M)$ scans to combine
- Multiway approaches are benifitial when the memory bandwidth is an issue!

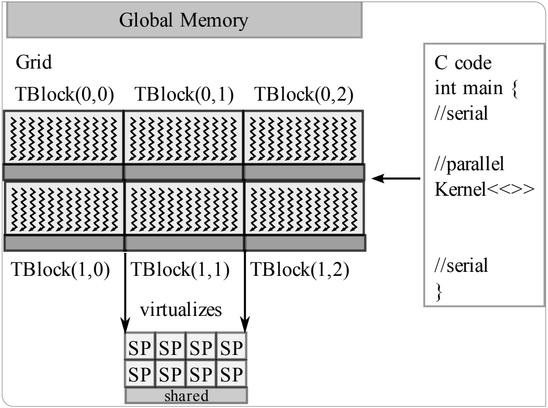
Fakultät für Informatik
Institut für Theoretische Informatik

NVidia Tesla Architecture



- 30 Streaming Processors (SM) $\times$ 8 Scalar Processors (SP) each
- overall 240 physical cores
- 16KB shared memory per SM similar to CPU L1 cache
- 4GB global device memory

Computing Unified Device Architecture Model



- Similar to SPMD (single-program multiple-data) model
  - block of concurrent threads execute a scalar sequential program, a kernel
  - thread blocks constitute a grid

Performance Guidelines

- General pattern in GPU algorithm design
  - decompose the problem into many data-independent sub-problems
  - solve sub-problems by blocks of cooperative parallel threads
- Performance Guidelines
  - conditional branching
    - follow the same execution path
  - shared memory
    - exploit fast on-chip memory
  - coalesced global memory operations
    - load/store requests to the same memory block
    $\rightsquigarrow$ fewer memory accesses

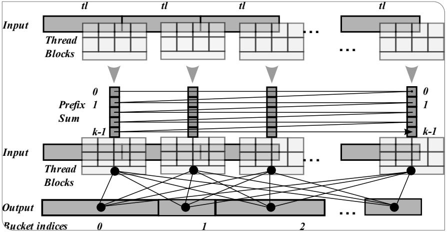
Algorithm Overview

```
SampleSort(e = ⟨e_1, ..., e_n⟩, k)
begin
    if n < M then  return SmallSort(e)
    choose a random sample S = S_1, ..., S_{ak-1} of e
    Sort(S)
    ⟨s_0, s_1, ..., s_k⟩ = ⟨-∞, S_a, ..., S_{a(k-1)}, ∞⟩
    for 1 ≤ i ≤ n do
        find j ∈ {1, ..., k}, such that s_{j-1} ≤ e_i ≤ s_j
        place e_i in bucket b_j
        return Concatenate(SampleSort(b_1, k), ..., SampleSort(b_k, k))
    end
end
```

Algorithm 1: Serial Sample Sort

High Level GPU Algorithm Design

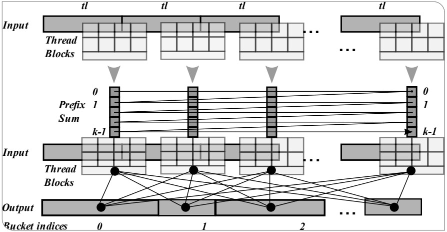
Karlsruher Institut für Technologie



- Phase 1. Choose splitters
- Phase 2. Each of p TB:
  - computes its elements bucket indices $id, 0 \le id \le k-1$
  - stores the bucket sizes in DRAM
- Phase 3. Prefix sum over the $k \times p$ table $\rightsquigarrow$ global offsets
- Phase 4.
  - as in Phase 2 $\rightsquigarrow$ local offsets
  - local + global offsets $\rightsquigarrow$ final positions

- Parameters:
  - distribution degree $k = 128$
  - threads per block $t = 256$
  - elements per thread $l = 8$
  - number of blocks $p = n/(t \cdot l)$

High Level GPU Algorithm Design



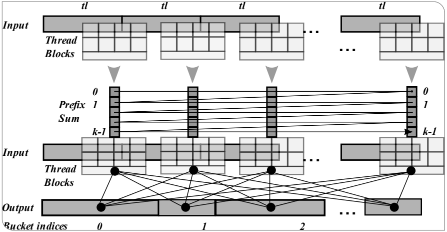
- Parameters:
  - distribution degree $k = 128$
  - threads per block $t = 256$
  - elements per thread $l = 8$
  - number of blocks $p = n/(t \cdot l)$

- Phase 1. Choose splitters
- Phase 2. Each of p TB:
  - computes its elements bucket indices $id, 0 \le id \le k-1$
  - stores the bucket sizes in DRAM
- Phase 3. Prefix sum over the $k \times p$ table $\rightsquigarrow$ global offsets
- Phase 4.
  - as in Phase 2 $\rightsquigarrow$ local offsets
  - local + global offsets $\rightsquigarrow$ final positions

 Vitaly Osipov:

Fakultät für Informatik
Institut für Theoretische Informatik

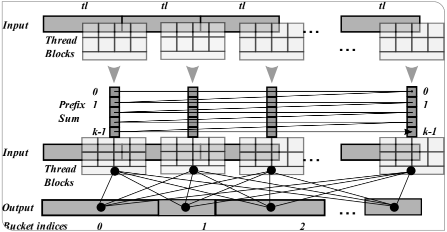
High Level GPU Algorithm Design



- Parameters:
  - distribution degree $k = 128$
  - threads per block $t = 256$
  - elements per thread $l = 8$
  - number of blocks $p = n/(t \cdot l)$

- Phase 1. Choose splitters
- Phase 2. Each of p TB:
  - computes its elements bucket indices $id, 0 \leq id \leq k-1$
  - stores the bucket sizes in DRAM
- Phase 3. Prefix sum over the $k \times p$ table $\rightsquigarrow$ global offsets
- Phase 4.
  - as in Phase 2 $\rightsquigarrow$ local offsets
  - local + global offsets $\rightsquigarrow$ final positions

High Level GPU Algorithm Design



- Parameters:
  - distribution degree $k = 128$
  - threads per block $t = 256$
  - elements per thread $l = 8$
  - number of blocks $p = n/(t \cdot l)$

- Phase 1. Choose splitters
- Phase 2. Each of p TB:
  - computes its elements bucket indices $id, 0 \leq id \leq k-1$
  - stores the bucket sizes in DRAM
- Phase 3. Prefix sum over the $k \times p$ table $\rightsquigarrow$ global offsets
- Phase 4.
  - as in Phase 2 $\rightsquigarrow$ local offsets
  - local + global offsets $\rightsquigarrow$ final positions

Fakultät für Informatik
Institut für Theoretische Informatik

Flavor of Implementation Details

computing element bucket indices

$bt = \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8} \ldots \rangle$

`TraverseTree(`e_i`)`

begin

$j := 1$

`// go left or right?`

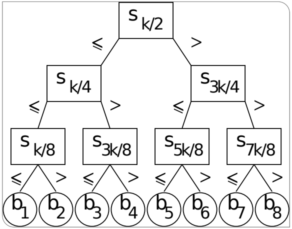
repeat $\log k$ times

$j := 2j + (e_i > bt[j])$

`// bucket index`

$j := j - k + 1$

end



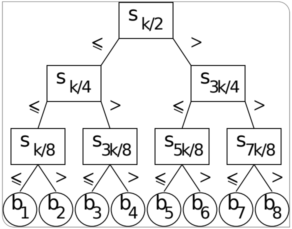
- Srore the tree in fast shared memory
- Use predicated instructions ⇝ no path divergence
- Unroll the loop

Flavor of Implementation Details

computing element bucket indices

$bt = \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8} \ldots \rangle$

`TraverseTree`(e_i)
begin
  $j := 1$
  `// go left or right?`
  repeat log k times
  $j := 2j + (e_i > bt[j])$
  `// bucket index`
  $j := j - k + 1$
end



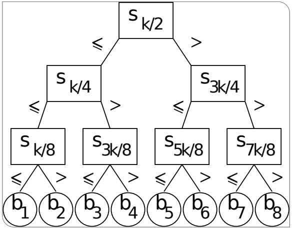
- Srore the tree in fast shared memory
- Use predicated instructions ⇝ no path divergence
- Unroll the loop

Flavor of Implementation Details

computing element bucket indices

$bt = \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8} \ldots \rangle$

```
TraverseTree(e_i)
begin
    j := 1
    // go left or right?
    repeat log k times
    j := 2j + (e_i > bt[j])
    // bucket index
    j := j − k + 1
end
```



- Srore the tree in fast shared memory
- Use predicated instructions ⇝ no path divergence
- Unroll the loop

Vitaly Osipov:
GPU Sample Sort

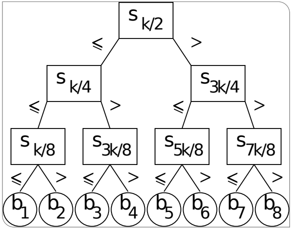
Fakultät für Informatik
Institut für Theoretische Informatik

Flavor of Implementation Details

computing element bucket indices

$bt = \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8} \ldots \rangle$

```
TraverseTree(e_i)
begin
    j := 1
    // go left or right?
    repeat log k times
    j := 2j + (e_i > bt[j])
    // bucket index
    j := j − k + 1
end
```



- Srore the tree in fast shared memory
- Use predicated instructions ⇝ no path divergence
- Unroll the loop

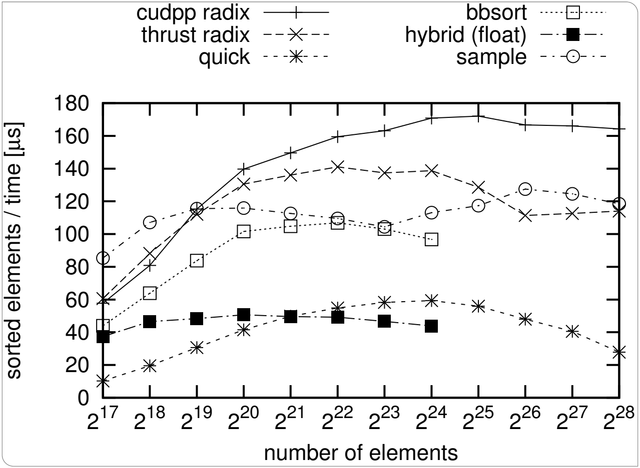
Experimental Evaluation

Karlsruher Institut für Technologie

- NVidia Tesla C1060
  - 30 SMs x 8 SPs = 240 cores
  - 4GB RAM
- Data types
  - 32- and 64-bit integers
  - key-value pairs
- Distributions
  - Uniform
  - Gausian
  - Bucket Sorted
  - Staggered
  - Deterministic Duplicates
- GPU sorting Algorithms
  - CUDPP and THRUST radix sort
  - THRUST merge sort
  - quicksort
  - hybrid sort
  - bbsort

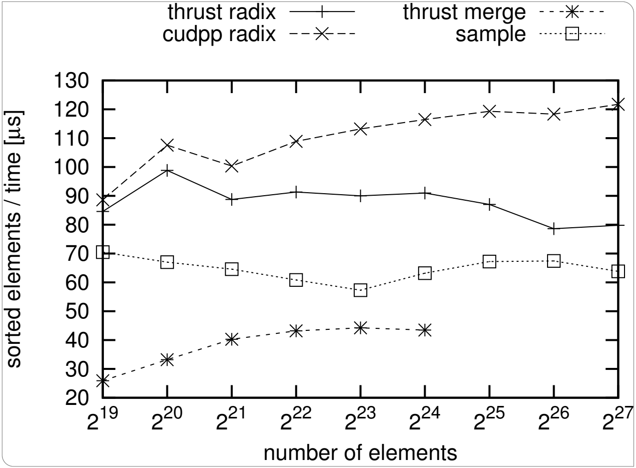
Experimental Evaluation

Uniform 32-bit integers



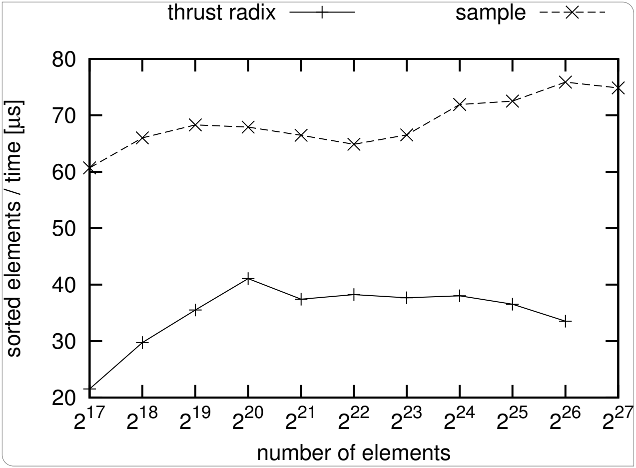
Experimental Evaluation

Uniform key-value pairs



Experimental Evaluation

Uniform 64-bit integers



Future Trends

Fermi architecture

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

- What about memory bandwidth? No significant improvements?
- multi-way approaches are likely to be even more beneficial
- multi-way merge sort?

Thank you!