

GPU Sample Sort

Nikolaj Leischner, *Vitaly Osipov*, Peter Sanders

Institut für Theoretische Informatik - Algorithmik II



- 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 intermediate 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 intermediate 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 beneficial 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 intermediate 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 intermediate 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
- **Multway approaches** are beneficial when the memory bandwidth is an issue!

Introduction

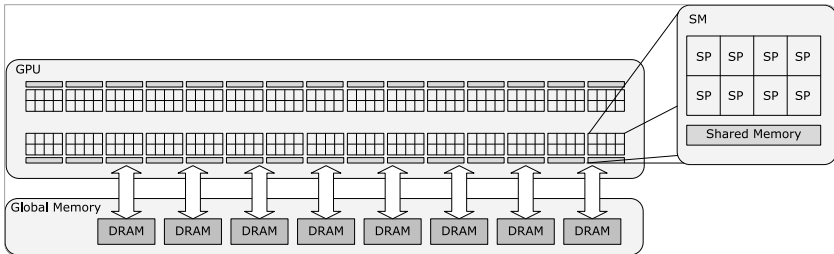
multi-way sorting algorithms

- **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 intermediate 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 intermediate 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
- **Multway approaches** are beneficial when the memory bandwidth is an issue!

Introduction

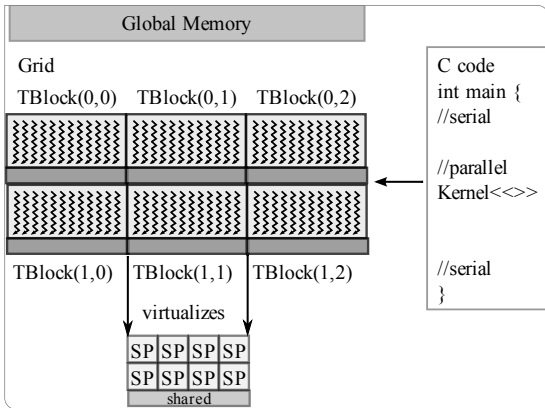
multi-way sorting algorithms

- **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 intermediate 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 intermediate 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
- **Multway approaches** are beneficial when the memory bandwidth is an issue!



- 30 Streaming Processors (SM) × 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

- 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
 - ↔ fewer memory accesses

Algorithm Overview

SampleSort($e = \langle e_1, \dots, e_n \rangle, k$)

begin

if $n < M$ **then return** SmallSort(e)

choose a random sample $S = S_1, \dots, S_{ak-1}$ of e

Sort(S)

$\langle s_0, s_1, \dots, s_k \rangle = \langle -\infty, S_a, \dots, S_{a(k-1)}, \infty \rangle$

for $1 \leq i \leq n$ **do**

find $j \in \{1, \dots, k\}$, such that $s_{j-1} \leq e_i \leq s_j$

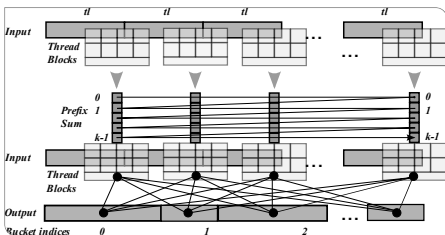
place e_i in bucket b_j

return Concatenate(SampleSort(b_1, k), ..., SampleSort(b_k, k))

end

end

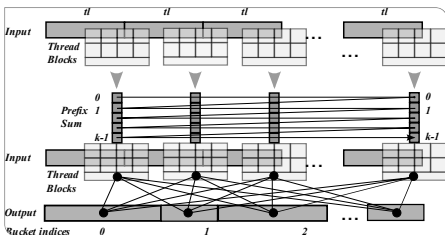
Algorithm 1: Serial Sample Sort



Parameters:

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

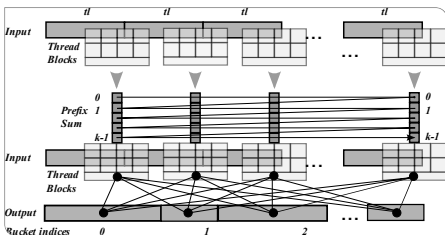
- Phase 1. Choose splitters
- Phase 2. Each of p TB:
 - computes its elements
 - bucket indices $id, 0 \leq id \leq k - 1$
 - stores the bucket sizes in DRAM
- Phase 3. Prefix sum over the $k \times p$ table \rightsquigarrow global offsets
- Phase 4.
 - as in Phase 2 \rightsquigarrow local offsets
 -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



Parameters:

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

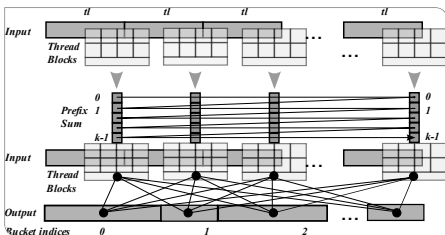
- Phase 1. Choose splitters
- Phase 2. Each of p TB:
 - computes its elements **bucket indices**
 $id, 0 \leq id \leq k - 1$
 - stores the **bucket sizes** in DRAM
- Phase 3. Prefix sum over the $k \times p$ table \rightsquigarrow **global offsets**
- Phase 4.
 - as in Phase 2 \rightsquigarrow **local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



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**



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

Flavor of Implementation Details

computing element bucket indices

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

TraverseTree(e_j)

begin

$j := 1$

// go left or right?

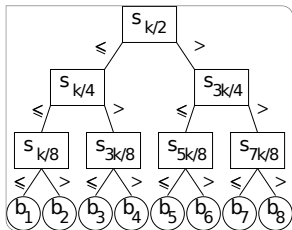
repeat $\log k$ times

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

// bucket index

$j := j - k + 1$

end



- Store the tree in fast shared memory
- Use predicated instructions \rightsquigarrow no path divergence
- Unroll the loop

Flavor of Implementation Details

computing element bucket indices

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

TraverseTree(e_j)

begin

$j := 1$

// go left or right?

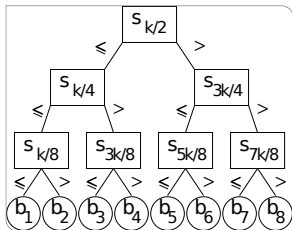
repeat $\log k$ times

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

// bucket index

$j := j - k + 1$

end



- Store the tree in **fast shared memory**
- Use predicated instructions \rightsquigarrow **no path divergence**
- Unroll the loop

Flavor of Implementation Details

computing element bucket indices

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

TraverseTree(e_j)

begin

$j := 1$

// go left or right?

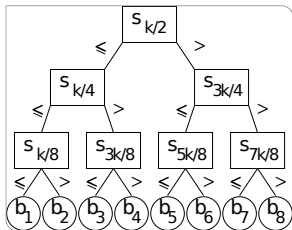
repeat $\log k$ times

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

// bucket index

$j := j - k + 1$

end



- Store the tree in **fast shared memory**
- Use predicated instructions \rightsquigarrow **no path divergence**
- Unroll the loop

Flavor of Implementation Details

computing element bucket indices

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

TraverseTree(e_j)

begin

$j := 1$

// go left or right?

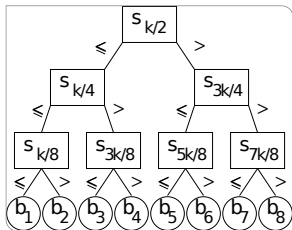
repeat $\log k$ times

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

// bucket index

$j := j - k + 1$

end

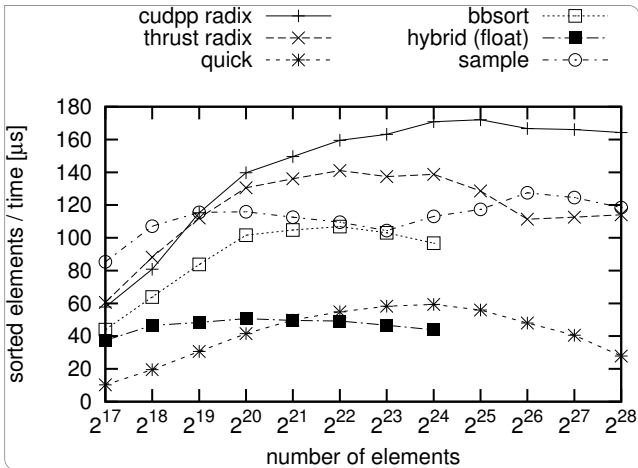


- Store the tree in **fast shared memory**
- Use predicated instructions \rightsquigarrow **no path divergence**
- Unroll the loop

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

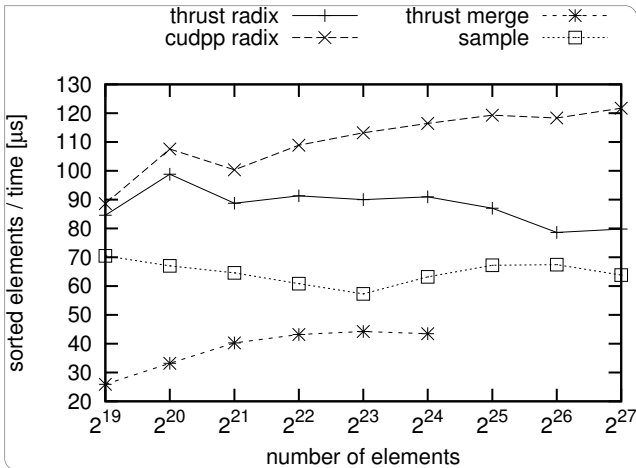
Experimental Evaluation

Uniform 32-bit integers



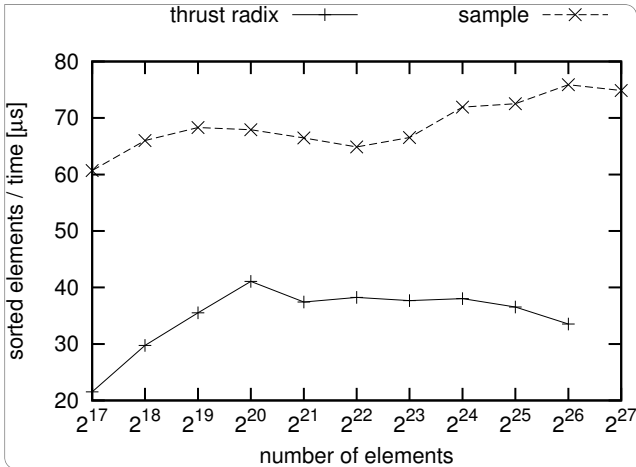
Experimental Evaluation

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!