GPU Sample Sort
Nikolaj Leischner, Vitaly Osipov, Peter Sanders
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 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 $\leadsto \log_2 (n/M)$ scans to partition the input
  - two-way merge sort $\leadsto \log_2 (n/M)$ scans to combine intermediate results
- Multi-way approaches:
  - $k$-way distribution - sample sort $\leadsto$ only $\log_k (n/M)$ scans to partition
  - $k$-way merge sort $\leadsto$ 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 $\sim \log_2 \left( \frac{n}{M} \right)$ scans to partition the input
  - two-way merge sort $\sim \log_2 \left( \frac{n}{M} \right)$ scans to combine intermediate results
- Multi-way approaches:
  - $k$-way distribution - sample sort $\sim$ only $\log_k \left( \frac{n}{M} \right)$ scans to partition
  - $k$-way merge sort $\sim$ only $\log_k \left( \frac{n}{M} \right)$ 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 $\leadsto \log_2 (n/M)$ scans to partition the input
  - two-way merge sort $\leadsto \log_2 (n/M)$ scans to combine intermediate results
- Multi-way approaches:
  - $k$-way distribution - sample sort $\leadsto$ only $\log_k (n/M)$ scans to partition
  - $k$-way merge sort $\leadsto$ only $\log_k (n/M)$ scans to combine
- Multiway approaches are beneficial when the memory bandwidth is an issue!
- 30 Streaming Processors (SM) \times 8 \text{ Scalar Processors (SP) each}
- \text{overall 240 physical cores}
- \text{16KB shared memory per SM similar to CPU L1 cache}
- \text{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
    ~ more efficient memory access
SampleSort\((e = \langle e_1, \ldots, e_n \rangle, k)\)
begin
  if \( n < M \) then return SmallSort\((e)\)
  choose a random sample \( S = S_1, \ldots, S_{ak-1} \) of \( e \)
  Sort\((S)\)
  \( \langle s_0, s_1, \ldots, s_k \rangle = \langle -\infty, S_a, \ldots, S_{a(k-1)}, \infty \rangle \)
  for \( 1 \leq i \leq n \) do
    find \( j \in \{1, \ldots, k\} \), such that \( s_{j-1} \leq e_i \leq s_j \)
    place \( e_i \) in bucket \( b_j \)
  return Concatenate(SampleSort\((b_1, k)\), \ldots, SampleSort\((b_k, k)\))
end

end

Algorithm 1: Serial Sample Sort
High Level GPU Algorithm Design

- **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 $\Rightarrow$ global offsets
- **Phase 4.**
  - as in Phase 2 $\Rightarrow$ local offsets
  - local + global offsets $\Rightarrow$ 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

- 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 $\mapsto$ global offsets
- Phase 4.
  - as in Phase 2 $\mapsto$ local offsets
  - local + global offsets $\mapsto$ 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 \(\leadsto\) global offsets

Phase 4.
- as in Phase 2 \(\leadsto\) local offsets
- local + global offsets \(\leadsto\) 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

- **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 $\Rightarrow$ global offsets
- **Phase 4.**
  - as in Phase 2 $\Rightarrow$ local offsets
  - local + global offsets $\Rightarrow$ final positions

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

computing element bucket indices

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

\bf{TraverseTree(}e_i\textbf{)}

\textbf{begin}
\begin{itemize}
  \item \hspace{1em} \texttt{j} := 1
  \item \hspace{1em} // go left or right?
  \item \hspace{1em} \textbf{repeat} \log k \textbf{times}
  \item \hspace{1em} \texttt{j} := 2j + (e_i > bt[j])
  \item \hspace{1em} // bucket index
  \item \hspace{1em} \texttt{j} := j - k + 1
\end{itemize}
\textbf{end}

\begin{itemize}
  \item Store the tree in fast shared memory
  \item Use predicated instructions \ freopen no path divergence
  \item Unroll the loop
\end{itemize}
Flavor of Implementation Details

computing element bucket indices

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

\text{TraverseTree(e)}

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

- Store the tree in fast shared memory
- Use predicated instructions \( \Rightarrow \) no path divergence
- Unroll the loop
Flavor of Implementation Details
computing element bucket indices

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

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

- Store the tree in fast shared memory
- Use predicated instructions \( \leadsto \) no path divergence
- Unroll the loop
Flavor of Implementation Details
computing element bucket indices

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

\text{TraverseTree}(e_i)
\begin{align*}
  j &:= 1 \\
  &\quad \text{// go left or right?} \\
  \text{repeat} \, \log k \, \text{times} \\
  j &:= 2j + (e_i > bt[j]) \\
  &\quad \text{// bucket index} \\
  j &:= j - k + 1 \\
\end{align*}

- Store the tree in fast shared memory
- Use predicated instructions $\leadsto$ no path divergence
- Unroll the loop
Experimental Evaluation

- 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

![Graph showing performance comparison of different sorting algorithms]

- cudpp radix
- thrust radix
- quick
- hybrid (float)
- sample

Vitaly Osipov: GPU Sample Sort
Experimental Evaluation

Uniform key-value pairs

thrust radix ——- thrust merge —-*

cudpp radix ——* —— sample ——•

sorted elements / time [µs]

number of elements

2^{19} 2^{20} 2^{21} 2^{22} 2^{23} 2^{24} 2^{25} 2^{26} 2^{27}
Experimental Evaluation
Uniform 64-bit integers

Vitaly Osipov: GPU Sample Sort

Karlsruher Institut für Technologie

sorted elements / time [µs]

number of elements

thrust radix —— sample

2^17 2^18 2^19 2^20 2^21 2^22 2^23 2^24 2^25 2^26 2^27
### Future Trends

#### Fermi architecture

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

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