Fast Segmented Sort on GPUs

Kaixi Hou†, Weifeng Liu‡, Hao Wang†, Wu-chun Feng†

†{kaixihou, hwang121, wfeng}@vt.edu ‡ weifeng.liu@nbi.ku.dk
Segmented Sort (SegSort)

- Perform a segment-by-segment sort on a given array composed of multiple segments

\[
\text{seg_ptr} = [0\ 3\ 5\ 7]
\]

\[
\text{input} = [4\ 1\ 2\ 11\ 8\ 1\ 6\ 5]
\]

\[
\text{output} = [1\ 2\ 4\ 8\ 11\ 1\ 6\ 5]
\]

Segmented sort
Why Segmented Sort?

- Many applications need to process (e.g., sort) a large amount of independent arrays, due to: (1) dataset properties, (2) algorithm characteristics.

Segment statistics from squaring one matrix in SpGEMM*

Segment statistics from 1st iteration in SAC*

* SpGEMM: Sparse General Matrix-Matrix Multiplication; SAC: Suffix Array Construction
Why Segmented Sort?

- Many applications need to process (e.g., sort) a large amount of independent arrays, due to: (1) dataset properties, (2) algorithm characteristics

We need an efficient way to deal with the large amount of independent short arrays.

Segment statistics from squaring one matrix in SpGEMM*

Segment statistics from 1st iteration in SAC*

* SpGEMM: Sparse General Matrix-Matrix Multiplication; SAC: Suffix Array Construction
Existing Segmented Sort

- Global sort has received much more fanfare!
- Many tools are evolved from global sort; however, there are also problems
  - Problem 1: **Time complexity**

\[
\text{The complexity of this segsort is } O\left(\frac{N}{n} \log n\right) \approx O(N \log n) *
\]

\[
\text{The complexity of the global sort is } O(N \log N) *
\]

* For generality, the sorting algorithms are all comparison-based.
Existing Segmented Sort

- Global sort has received much more fanfare!
- Many tools are evolved from global sort; however, there are also problems
  - Problem 1: Time complexity
  - Problem 2: Runtime boundary checking overhead

![Diagram](image)

Every thread needs to check boundary of segments for every comparison.

Some SegSort needs to perform runtime boundary checking, causing additional overhead, e.g., segsort from *modernGPU*.
Existing Segmented Sort

• Global sort has received much more fanfare!
• Many tools are evolved from global sort; however, there are also problems
  – Problem 1: Time complexity
  – Problem 2: Runtime boundary checking overhead
  – Problem 3: **Underutilized resources**

Some SegSort simply assigns each segment to each thread block, leading to idle resources, e.g., segsort from *CUB*

Many threads might be idle, especially when the segments are generally short.
We propose an adaptive segmented sort mechanism for GPUs: (1) differentiated methods for different segments, (2) an algorithm supporting variable data-thread binding and thread communication.
Outline

• Introduction
• Motivation

• Our Method
  – GPU SegSort Mechanism
  – GPU Register-based Sort
  – Other Techniques & Opt.

• Evaluation
  – Kernel Performance
  – Kernel in Real Applications
Adaptive GPU SegSort Mechanism

- Overview of our proposed GPU SegSort design

Segments (seg_ptr & input)

- t = thread
- w = warp
- b = block
Adaptive GPU SegSort Mechanism

- Overview of our proposed (GP) SegSort design

  - Hierarchical binning (register/smem/gmem levels)

Segments ($\text{seg}_\text{ptr}$)  

unit-bin  
warp-bin  
block-bin  
grid-bin

$\text{t} = \text{thread}$  
$\text{w} = \text{warp}$  
$\text{b} = \text{block}$
Adaptive GPU SegSort Mechanism

• Overview of our proposed GPU SegSort design

Segments ($seg\_ptr \& input$)

- unit-bin
- warp-bin
- block-bin
- grid-bin

Global memory: sorted segments ($output$)

reg-sort

smem-merge
Adaptive GPU SegSort Mechanism

• Overview of our proposed GPU SegSort design

Segments ($seg_{ptr} & input$)

- Directly copy segments to the result in global memory

Global memory: sorted segments ($output$)

- reg-sort
- smem-merge

$t = \text{thread}$

$w = \text{warp}$

$b = \text{block}$
Adaptive GPU SegSort Mechanism

• Overview of our proposed GPU SegSort design

Segments ($seg_{ptr}$ & input)

- Global memory: sorted segments ($output$)
  - Only use registers as “cache”
  - Data-thread binding/exchange
  - Memory access optimization

- t = thread
  w = warp
  b = block

- Unit-bin
- Warp-bin
- Block-bin
- Grid-bin

- striped-write
- reg-sort
- smem-merge
Adaptive GPU SegSort Mechanism

- Overview of our proposed GPU SegSort design

Segments \((seg\_ptr \& \text{input})\)

- Use registers + smem as “cache”
- **MergePath** algorithm for load balance

Global memory: sorted segments \((output)\)

- **t** = thread
- **w** = warp
- **b** = block
Adaptive GPU SegSort Mechanism

- Overview of our proposed GPU SegSort design

Segments \((\text{seg\_ptr \& input})\)

- \(t = \text{thread}\)
- \(w = \text{warp}\)
- \(b = \text{block}\)

Global memory: sorted segments \((\text{output})\)
GPU Register-based Sort

- **Sorting networks** usually serve as building blocks of efficient parallel sort
- How to bind the data items (operands) to different threads?
GPU Register-based Sort

- **Sorting networks** usually serve as building blocks of efficient parallel sort.
- How to bind the data items (operands) to different threads?

Both need to figure out data exchange patterns among registers and threads.

- Each thread uses too many registers (**high register pressure**).
- Each comparison is performed twice (**wasted computing resources**).
GPU Register-based Sort

- Propose a general way to solve the data-thread binding problem at GPU register level
- Primitive pattern

```
_exch_primitive(rg0, rg1, tmask, swbit)
```

Implementation Details

1. `_shuf_xor(rg1, 0x1); // Shuffle data in rg1`
2. `cmp_swp(rg0, rg1); // Compare data of rg0 & rg1 locally`
3. `if( bfe(tid, 0) ) swp(rg0, rg1); // Swap data of rg0 & rg1 if 0 bit of tid is set`
4. `_shuf_xor(rg1, 0x1); // Shuffle data in rg1`

Two data items are bound to each thread
Tells which thread swaps registers
Tells how threads communicate
Other patterns, then, can be solved by transformation and the primitive patterns

**Intersecting Pattern**

Any number of data items are bound to each thread

Tells which thread swaps registers

Tells how threads communicate

Applying primitive patterns on related pairs

Transforming via swapping

if(bfe(tid, swbit))
swp(r0, rgk-2)
swp(r1, rgk-1)
...
• Other patterns, then, can be solved by transformation and the primitive patterns

• **Parallel Pattern**

\[ \text{exch\_paral}(rg_0, rg_1, ..., rg_{k-1}, tmask, swbit) \]

Any number of data items are bound to each thread

Tells which thread swaps registers

Tells how threads communicate

Transferring via swapping

Applying **primitive patterns** on related pairs

if(bfe(tid, swbit))

swp(rg0,rg1)

swp(rg2,rg3)

...
GPU Register-based Sort

- Also, we can solve patterns without thread communication
  - “Communication” only occurs between registers

- **Local Pattern**
  
  ```
  _exch_local(rg0, rg1, ..., rgk-1, rmask)
  ```

  Any number of data items are bound to each thread
  
  Tells how registers compare with each other
• Represent the sorting network by using our generalized patterns

\[
\text{reg_sort(data items=8, thread num=4)}
\]

Read our paper and see more details of (1) how to automatically decide which patterns to use, (2) how to order the patterns, (3) how to compute the parameters (e.g., tmask)
Other Techniques & Optimizations

- A hierarchical binning
  - Using warp vote function \_\_balloc() and \_\_popc() at warp level
  - Using shared memory at thread-block level
- Better locality by optimizing access pattern
  - Transforming striped write to coalesced memory access
• A hierarchical binning
  – Using warp vote function \_\_ballo\_\_ and \_\_popc\_\_ at warp level
  – Using shared memory at thread-block level

• Better locality by optimizing access pattern
  – Transforming striped write to coalesced memory access
Other Techniques & Optimizations

- A hierarchical binning
  - Using warp vote function __balloc() and __popc() at warp level
  - Using shared memory at thread-block level

- Better locality by optimizing access pattern
  - Transforming striped write to coalesced memory access

<table>
<thead>
<tr>
<th>Input</th>
<th>Stride=1</th>
<th>Stride=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>swap</td>
<td>swap</td>
</tr>
<tr>
<td>t1</td>
<td>exchange</td>
<td>exchange</td>
</tr>
<tr>
<td>t2</td>
<td>swap</td>
<td>swap</td>
</tr>
<tr>
<td>t3</td>
<td>shuf_xor(rg1,0x1)</td>
<td>shuf_xor(rg1,0x2)</td>
</tr>
</tbody>
</table>

Striped-access
Other Techniques & Optimizations

- A hierarchical binning
  - Using warp vote function __balloc() and __popc() at warp level
  - Using shared memory at thread-block level

- Better locality by optimizing access pattern
  - Transforming striped write to coalesced memory access
Other Techniques & Optimizations

- A hierarchical binning
  - Using warp vote function `__balloc()` and `__popc()` at warp level
  - Using shared memory at thread-block level

- Better locality by optimizing access pattern
  - Transforming striped write to coalesced memory access

- Shared memory based merge solution
  - MergePath algorithm [\`12] for load balance
Outline

• Introduction
• Motivation

• Our Method
  – GPU SegSort Mechanism
  – GPU Register-based Sort
  – Other Techniques & Opt.

• Evaluation
  – Kernel Performance
  – Kernel in Real Applications
Experiment Platforms

- **nVidia Tesla K80 (Kepler-GK210)**, 2496 CUDA cores @ 824 MHz, 240 GB/s bandwidth
- **nVidia TitanX (Pascal-GP102)**, 3584 CUDA cores @ 1531 MHz, 480 GB/s bandwidth
- We compare our **SegSort** to other tools from libraries of
  a. ModernGPU v.2.0 (boundary checking, global sort based)
  b. CUSP* v.0.5.0 (global sort based)
  c. CUB v.1.6.4 (segment per block)
    - Generating datasets to mimic different segment distributions
- We compare **SAC** and **SpGEMM** optimized by our **SegSort** to
  a. cuDPP v.2.3 for SAC
  b. cuSPARSE [\`16], CUSP [\`14], bhSPARSE [\`14] for SpGEMM
    - Using real input datasets from NCBI library and UF matrix collection

* CUSP performs segmented sort by using THRUST sort twice. We extract this as a stand-alone function.
Kernel Performance Tuning

- Binding different number of data items to threads (\textit{reg\_sort})

**Kepler GPU**

**Striped access**
- \texttt{blk\_064}
- \texttt{blk\_128}
- \texttt{blk\_256}
- \texttt{blk\_512}

**Coalesced access**
- \texttt{blk\_064}
- \texttt{blk\_128}
- \texttt{blk\_256}
- \texttt{blk\_512}

Throughput (10^9 pairs/s) vs. Pairs per thread for different thread counts (2, 4, 8, 16, 32) on Kepler GPU.
Kernel Performance Tuning

- Binding different number of data items to threads (reg_sort)

**Kepler GPU**

- Striped access
- Coalesced access

![Graph showing Kernel Performance Tuning](chart.png)
Kernel Performance Tuning

- Binding different number of data items to threads (reg_sort)

Striped access

Coalesced access

Pascal GPU

Throughput (10^9 pairs/s)

Pairs per thread

8 threads

16 threads

32 threads

Throughput (10^9 pairs/s)

Pairs per thread
Kernel Performance Tuning

- Binding different number of data items to threads (reg_sort)

**Pascal GPU**

**Striped access**
- blk_064
- blk_128
- blk_256
- blk_512

**Coalesced access**
- blk_064
- blk_128
- blk_256
- blk_512

Graph showing throughput (10e9 pairs/s) vs. pairs per thread for different block sizes and thread counts.
SegSort Performance

- Fixing total data size w/ variable segment number and size

Our SegSort is proficient in solving a large amount of segments, achieving an average of 3.2x speedups over the better performed baseline mgpu-segsort on Pascal.
SegSort Performance

- Fixing total data size \( w/ \) variable segment number and size

- Our SegSort is proficient in solving a large amount of segments, achieving an average of 3.2x speedups over the better performed baseline mgpu-segsort on Pascal

- The performance of SegSorts, evolved from global sort, is more affected by the total array size
SegSort Performance

- Fixing total data size with segments of power-law distribution.

![Speedups vs. CUB vs. CUSP vs. ModernGPU](chart)

Kepler GPU

Pascal GPU

Max Segment Size

Larger max length

More small segments

Alpha

[vS. CUB vs. CUSP vs. ModernGPU](chart)
SegSort Performance

- Fixing total data size with segments of power-law distribution.

**Speedups**

- vs. CUB
- vs. CUSP
- vs. ModernGPU

Kepler GPU

Pascal GPU

- CUB is limited by processing only 65536 segments
- We can achieve up to 87x speedups
SegSort Performance

- Fixing total data size with segments of power-law distribution.

### Speedups

#### vs. CUB

- Kepler GPU
- Pascal GPU

#### vs. CUSP

#### vs. ModernGPU

- Larger max length
- More small segments
- Alpha
- Max Segment Size
SegSort Performance

• Fixing total data size w/ segments of power-law distribut.

Speedups

<table>
<thead>
<tr>
<th></th>
<th>vs. CUB</th>
<th>vs. CUSP</th>
<th>vs. ModernGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler GPU</td>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>Pascal GPU</td>
<td><img src="image4" alt="Graph" /></td>
<td><img src="image5" alt="Graph" /></td>
<td><img src="image6" alt="Graph" /></td>
</tr>
</tbody>
</table>

- CUSP conducts global sort twice
- We can achieve up to 17x speedups
SegSort Performance

- Fixing total data size w/ segments of power-law distribut.

Speedups

vs. CUB

vs. CUSP

vs. ModernGPU

Kepler GPU

Pascal GPU

Alpha

More small segments

Larger max length

Max Segment Size
SegSort Performance

- Fixing total data size with segments of power-law distribution.

**Speedups**

- vs. CUB
- vs. CUSP
- vs. ModernGPU

Kepler GPU

Pascal GPU

- ModernGPU has runtime boundary checking overhead
- We can achieve up to 4x speedups
• **Suffix Array**: store lexicographically sorted indices of all suffixes of a given sequence

• Our method is based on the *prefix doubling* algorithm [`93]
  – Deducing the orders of $2^h$ strings from the calculated orders of $h$ strings
SegSort in Real-world Applications

Sparse Matrix-Matrix Multiplication (Sp) by its indices of rows and columns

- Our method is using the Expansion, Seg-Sorting and Compression (ESSC) algorithm [12]
  - Sorting an intermediate sparse matrix $C$ by its indices of rows and columns
SegSort in Real-world Applications

Sparse Matrix-Matrix Multiplication (Sp by its indices of rows and columns

Our method is using the Expansion, Seg-Sorting and
Compression (ESSC) algorithm [12] – Sorting an intermediate sparse matrix by its indices of rows and columns

Kepler GPU

Pascal GPU

Execution Time (ms)

Matrix
Conclusion

• We identified the importance of segmented sort on various applications, and proposed efficient approaches on GPUs
• Our GPU segmented sort method outperforms other state-of-the-art approaches in libraries of CUB, CUSP, ModernGPU
• We can see that the capacity of registers is important for segmented sort in modern GPUs
• Please visit our GIT repo https://github.com/vtsynergy

Thank you! Email to kaixihou@vt.edu
More from synergy.cs.vt.edu