SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures

Hiroshi Inoue†‡ and Kenjiro Taura‡
† IBM Research – Tokyo
‡ University of Tokyo
Our Goal

- Develop a fast algorithm for sorting an array of structures

Our approach:
- To use SIMD-based multiway mergesort as the basis
- To exploit SIMD instructions to accelerate mergesort kernel
- To avoid random memory accesses that cause excessive cache misses
Outline

✓ Goal

- Background: SIMD multiway mergesort for integers
- Performance problems in existing approaches
- Our approach
- Performance results
Background:
Mergesort with SIMD for sorting integers
[Inoue et al. PACT 2007, Chhugani et al. PVLDB 2008 etc.]

- Merge operation for 32-bit or 64-bit integers can be efficiently implemented with SIMD by:
  - merging multiple values in vector registers using SIMD min and max instructions (i.e. without conditional branches)
  - integrating the in-register vector merge into usual comparison-based merge operation

- SIMD min and max instructions can accelerate sorting by
  - parallelizing comparisons
  - avoiding unpredictable conditional branches
Background:
SIMD-based merge for values in two vector registers

Input
two vector registers contain four presorted values in each

Output
eight values in two vector registers are now sorted

(example of odd-even merge)
Background:
Integrating Odd-even merge into usual merge

sorted array 1

| 1 | 4 | 7 | 9 | 10 | 11 | 12 | 14 | 21 | 23 |

sorted array 2

| 2 | 3 | 5 | 6 | 8 | 13 | 15 | 16 | 17 | 18 |

sorted
Background: Integrating Odd-even merge into usual merge

sorted array 1

1 4 7 9 10 11 12 14 21 23

sorted array 2

2 3 5 6 8 13 15 16 17 18

vector registers

1 4 7 9 2 3 5 6

register-level merge
Background:
Integrating Odd-even merge into usual merge

sorted array 1

1 4 7 9 10 11 12 14 21 23

sorted array 2

2 3 5 6 8 13 15 16 17 18

vector registers

1 2 3 4 5 6 7 9

sorted
Background:
Integrating Odd-even merge into usual merge

Sort array 1 and sort array 2 into vector registers. Use a scalar comparison to select array to load from. Output smaller four values as merged array.
Background: Integrating Odd-even merge into usual merge

sorted array 1: 1 4 7 9 10 11 12 14 21 23
sorted array 2: 2 3 5 6 8 13 15 16 17 18
vector registers: 8 13 15 16 5 6 7 9
merged array: 1 2 3 4

register-level merge

sorted
Background:
Integrating Odd-even merge into usual merge

sorted array 1

1 4 7 9 10 11 12 14 21 23

sorted array 2

2 3 5 6 8 13 15 17 18

vector registers

merged array

1 2 3 4 5 6 7 8

sorted
Background: Multiway mergesort

Standard (2-way) mergesort

Multiway (k-way) mergesort, here $k = 4$
Background: Multiway merge with SIMD

(8-way merge as an example. We used 32-way merge in actual implementation)
Outline

✓ Goal
✓ Background: SIMD multiway mergesort for integers
  ▪ Performance problems in existing approaches
  ▪ Our approach
  ▪ Performance results
Existing approach for sorting structures with SIMD

- For sorting structures with SIMD mergesort, a frequently used approach (*key-index approach*) is
  1. pack key and index for each record into an integer.
  2. sort the key-index pairs with SIMD, and then
  3. **rearrange** the records based on the sorted key-index pairs

-authored by:  
© 2015 IBM Corporation
Performance problems in existing approaches

- Existing approaches for sorting structures with SIMD instructions have performance problems
  - Key-index approach: SIMD friendly but not cache friendly due to random memory accesses
  - Direct approach: cache friendly but not SIMD friendly because the keys are not stored contiguously in memory

- Our approach takes the benefits of both approaches
Our approach: Rearranging at each multiway merge step

Key-index approach
- **unsorted** input
- encode
- merge integers
- rearrange
- cache unfriendly
- **sorted** output

Our approach
- **unsorted** input
- encode
- rearrange
- encode
- rearrange
- **sorted** output

Direct approach
- **unsorted** input
- merge structures
- SIMD unfriendly
- **sorted** output
Our approach: Rearranging at each multiway merge step

Key-index approach:
- Unsorted input
- Encode
- Merge integers
- Number of memory copy per record
  - Only once (random access)
- Rearrange
- Sorted output

Our approach:
- Unsorted input
- Encode
- Rearrange
- Sorted output

Direct approach:
- Unsorted input
- Merge structures
- At every 2-way merge step: \( \log_2(N) \)

Note:
- Multiple times: \( \log_k(N) \)
Our approach: Multiway Merge with SIMD

- Extract key and encode \{key, streamID\} pair into an integer value
- 2-way merge
- Rearrange records based on merged integer values
- Move only integers during merging
- Decode streamID and copy records
- Output stream

System memory

Input streams

Stage 1

Stage 2

Processor's cache memory
Our approach: Overall sorting scheme

Step 1: divide into small blocks that can fit in (L2) cache

Step 2: sort each block by vectorized combsort

Step 3: repeatedly execute multiway merge to merge all blocks

(sorted output)

(read the paper on vectorized combsort)
Performance Evaluations

- **System**
  - 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors
    - 2 sockets x 8 cores = 16 cores
    - using SSE instructions (128-bit SIMD)
  - 96 GB of system memory
  - Redhat Enterprise Linux 6.4, gcc-4.8.2

- We compared the performance of following algorithms
  - Multiway mergesort \( (k = 32 \text{ way}) \)
    - Our approach, key-index approach, direct approach
  - Radix sort, STL std::stable_sort, STL (unstable) std::sort
Performance for Sorting 512M 16-byte records with and without SIMD instructions

<table>
<thead>
<tr>
<th>Approach</th>
<th>Execution Time (sec) with SIMD</th>
<th>Execution Time (sec) without SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our approach</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>Key-index approach</td>
<td>40</td>
<td>70</td>
</tr>
<tr>
<td>Direct approach</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>Radix sort STL</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>STL stable_sort</td>
<td>120</td>
<td>130</td>
</tr>
<tr>
<td>STL sort (unstable)</td>
<td>140</td>
<td>150</td>
</tr>
</tbody>
</table>

- Our approach gained largest speed up from SIMD among three approaches.
- It achieved comparable performance to radix sort.
Performance Scalability with multiple cores

- Our approach achieved the best performance with multiple cores up to 16 cores.
- It showed better scalability than radix sort.

using 512M 16-byte records
Performance with various sizes of a record

Key-index approach performed well for very large record sizes due to smaller number of data copy using 16 million records on 1 core.
Summary

- We developed a new algorithm for sorting an array of structures, that enables
  - efficient use of SIMD instructions
  - efficient use of cache memory (no random accesses)

- Our new algorithm outperformed the key-index approaches in multiway mergesort especially when each record is smaller than a cache line

- It outperformed the radix sort for records larger than 16 byte and showed better scalability with multiple cores

Read the paper for more detail: efficient 4-wide SIMD exploitation, results with 10-byte ASCII key, sorting for variable-length strings
backup
Optimization techniques: partial key comparison

- To use 4-wide SIMD (32 bit x 4) instead of 2-wide SIMD (64 bit x 2)
  - Increasing data parallelism and giving the higher performance
  - Sorting based on a partial key
    - We confirm that the sorted order is correct using the entire key when rearranging records using scalar comparison
Sorting for variable-length strings

- 256M 12-byte to 20-byte records (16 bytes on average)

- 64M 12-byte to 84-byte records (48 bytes on average)

Our approach, Key-index approach, Direct approach, Radix sort

execution time (sec)

- faster
Performance with various numbers of records

![Graph showing performance with various numbers of records. The graph compares different sorting approaches, with 'Our approach' being the fastest. The x-axis represents the number of records, and the y-axis represents the execution time. The legend includes 'Our approach', 'Key-index approach', 'Direct approach', 'Radix sort', 'STL stable_sort', and 'STL sort (unstable)'. The graph indicates that 'Our approach' is the fastest.](image-url)
Effect of number of ways in multi-way merge

on 1 core

- Larger ways reduced path length but increased cache misses

on 16 cores

- Faster