AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors

Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu and Toshio Nakatani
IBM Tokyo Research Laboratory
Goal

Develop a fast sorting algorithm by exploiting the following features of today’s processors

► **SIMD instructions**
► **Multiple Cores** (Thread-level parallelism)

- Outperform existing algorithms, such as Quick sort, by exploiting SIMD instructions on a single thread
- Scale well by using thread-level parallelism of multiple cores
Benefit and limitation of SIMD instructions

**Benefits:** SIMD selection instructions (e.g. select minimum instruction) can accelerate sorting by
- parallelizing comparisons
- avoiding unpredictable conditional branches

**Limitation:** SIMD load / store instructions can be effective only when they access contiguous 128 bits of data (e.g. four 32-bit values) aligned on 128-bit boundary
SIMD instructions are effective for some existing sorting algorithms but slower than Quick sort

- Such as Bitonic merge sort, Odd-even merge sort, and GPUTeraSort [Govindaraju ’05]
- They are slower than Quick sort for sorting a large number \(N\) of elements
  - Their computational complexity is \(O(N (\log (N))^2)\)

A step of Bitonic merge sort

```
1 5 6 8 7 4 3 2
```

MIN MAX
Presentation Outline

- Motivation
- AA-sort: Our new sorting algorithm
- Experimental results
- Summary
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements

```
2  7  6  5  8  4  1  3
```

*starting to compare two elements with large distance*

`distance = 6`
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements

```
2 7 6 5 8 4 1 3
```

starting to compare two elements with large distance

distance = distance / 1.3 = 4

decreasing distance by dividing a constant called shrink factor (1.3) for each iteration
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements

Starting to compare two elements with large distance:

```
2 7 6 5 8 4 1 3
```

Decreasing distance by dividing a constant called shrink factor (1.3) for each iteration:

```
distance = distance / 1.3 = 3
```
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements

```
2 7 6 5 8 4 1 3
```

Starting to compare two elements with large distance

Decreasing distance by dividing a constant called shrink factor (1.3) for each iteration

Distance = distance / 1.3 = 2
What is Comb sort?

- Comb sort [Lacey '91] is an extension to Bubble sort
- It compares two *non-adjacent* elements

```
2  7  6  5  8  4  1  3
```

- starting to compare two elements with large distance
- distance = distance / 1.3 = 1
- decreasing distance by dividing a constant called shrink factor (1.3) for each iteration
- repeat until all data are sorted with distance = 1
- average complexity $N \log (N)$
Our technique to SIMDize the Comb sort

input data

| 5 | 3 | 2 | 11 | 9 | 4 | 12 | 1 | 10 | 6 | 8 | 7 |

SIMD instructions are not effective for Comb sort due to:
- unaligned memory accesses
- loop-carried dependencies

sorted output data

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

sorted
Our technique to SIMDize the Comb sort

Input data

| 5 | 3 | 2 | 11 | 9 | 4 | 12 | 1 | 10 | 6 | 8 | 7 |

Transposed order

| 1 | 4 | 7 | 10 | 2 | 5 | 8 | 11 | 3 | 6 | 9 | 12 |

SIMD instructions are effective for Comb sort into the Transposed order

Reorder after sorting

Sorted output data

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Sorted
Our technique to SIMDize the Comb sort

input data

\[
\begin{array}{cccccccccccc}
5 & 3 & 2 & 11 & 9 & 4 & 12 & 1 & 10 & 6 & 8 & 7
\end{array}
\]

Transposed order

\[
\begin{array}{cccccccccccc}
1 & 4 & 7 & 10 & 2 & 5 & 8 & 11 & 3 & 6 & 9 & 12
\end{array}
\]

sorted output data

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
\]

assume four elements in one vector

sorted
Our technique to SIMDize the Comb sort

input data

| 5 | 3 | 2 | 11 | 9 | 4 | 12 | 1 | 10 | 6 | 8 | 7 |

Transposed order

► unaligned access
► loop-carried dependency

sorted output data

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

► no unaligned access
► no loop-carried dependency

sorted
## Analysis of our SIMDized Comb sort

<table>
<thead>
<tr>
<th></th>
<th>our SIMDized Comb sort</th>
<th>naively SIMDized Comb sort</th>
<th>original (scalar) Comb sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of unpredictable conditional branches</td>
<td>almost 0</td>
<td>almost 0</td>
<td>$O(N \log(N))$</td>
</tr>
<tr>
<td>Number of unaligned memory accesses</td>
<td>almost 0</td>
<td>$O(N \log(N))$</td>
<td>N/A</td>
</tr>
<tr>
<td>Computational complexity</td>
<td>$O(N \log(N))$</td>
<td>$O(N \log(N))$</td>
<td>$O(N \log(N))$</td>
</tr>
<tr>
<td>reordering: $O(N)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overview of the AA-Sort

- It consists of two phases:
  - Phase 1: SIMDized Comb sort
  - Phase 2: SIMDized Merge sort
Our approach to SIMDize merge operations

- SIMD instructions are effective for Bitonic merge or Odd-even merge
- Their computational complexity are higher than usual merge operations

**Our solution**

- Integrate Odd-even merge into the usual merge operation to take advantage of SIMD instructions while keeping the computational complexity of usual merge operation
Odd-even merge for values in two vector registers

**Input**
Two vector registers contain four presorted values in each.

**Odd-even Merge**
One SIMD comparison and “shuffle” operations for each stage.
No conditional branches!

**Output**
Eight values in two vector registers are now sorted.
Our technique to integrate 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
Our technique to integrate 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

**Odd-even merge**
Our technique to integrate Odd-even merge into usual merge

sorted array 1

sorted array 2

vector registers
Our technique to integrate 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

merged array

1 2 3 4

5 6 7 9

use a scalar comparison to select array to load from

output smaller four values as merged array
Our technique to integrate Odd-even merge into usual merge

<table>
<thead>
<tr>
<th>sorted array 1</th>
<th>1</th>
<th>4</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>14</th>
<th>21</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array 2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>vector registers</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Odd-even merge

sorted array

merged array

1 | 2 | 3 | 4
Our technique to integrate 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
9 13 15 16

merged array
1 2 3 4 5 6 7 8

sorted
## Comparing merge operations

<table>
<thead>
<tr>
<th></th>
<th>our integrated merge operation</th>
<th>odd-even merge implemented with SIMD</th>
<th>usual (scalar) merge operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of unpredictable conditional branches</td>
<td>1 for every output <em>vector</em></td>
<td>0</td>
<td>1 for every output <em>element</em></td>
</tr>
<tr>
<td>Computational complexity</td>
<td>$O(N)$</td>
<td>$O(N \log(N))$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Parallelizing AA-sort among multiple threads

- **Phase 1**
  - Each thread sorts independent blocks using SIMDized Comb sort.
  - Each thread executes independent merge operations.

- **Phase 2**
  - Multiple threads cooperate on one merge operation.
Presentation Outline

- Motivation
- AA-sort: Our new sorting algorithm
- Experimental results
- Summary
Environment for Performance Evaluation

- We used two processors for performance evaluation
  - PowerPC 970 using VMX instructions (up to 4 cores)
  - Cell BE using SPE cores (up to 16 SPE cores)

- We compared the performance of four algorithms
  - AA-sort
  - GPUTeraSort [Govindaraju '05]
  - ESSL (IBM’s optimized library)
  - STL delivered with GCC (open source library)
Single-thread performance on PowerPC 970 for various input size

sorting 32-bit random integers on one core of PowerPC 970
Single-thread performance on PowerPC 970

sorting 32 M elements of random 32-bit integers on PowerPC 970

x1.7, x3.0, x3.3
Scalability with multiple cores on PowerPC 970

- Sorting 32 M elements of random 32-bit integers on PowerPC 970
- AA-sort vs. GPU TeraSort
- Relative throughput over AA-sort using 1 core
- Faster x2.6 by 4 cores
- Faster x2.1 by 4 cores
Scalability with multiple cores on Cell BE

- **AA-sort**
- **GPUTeraSort**

sorting 32 M elements of random 32-bit integers on Cell BE

- 12.2x by 16 cores
- 7.4x by 8 cores

Relative throughput over AA-sort using 1 core.
Summary

- We proposed a new sorting algorithm called AA-sort, which can take advantage of
  - SIMD instructions
  - Multiple Cores (thread-level parallelism)

- We evaluated the AA-sort on PowerPC 970 and Cell BE
  - Using only 1 core of PowerPC 970, the AA-sort outperformed
    - IBM’s ESSL by 1.7x
    - GPUTeraSort by 3.3x
  - On Cell BE, the AA-sort showed good scalability
    - 8 cores 7.4x
    - 16 cores 12.2x
Thank you for your attention!
SIMD instructions are not effective for Quick sort

- SIMD instructions are *NOT* effective for Quick sort, which requires element-wise and unaligned memory accesses

A step of Quick sort (pivot = 7)

| 5 | 3 | 2 | 11 | 9 | 4 | 12 | 1 | 10 | 6 | 8 | 13 |

search an element larger than the pivot

swap

search an element smaller than the pivot

element-wise and unaligned memory accesses
Transposed order

- To see the values to sort as a two dimensional array

adjacent values are stored in one vector

adjacent values are stored in same slot of adjacent vectors