ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors

Kaixi Hou, Hao Wang, Wu-chun Feng
{kaixihou, hwang121, wfeng}@vt.edu
Why sorting?

- Sorting used as an important primitive in many applications
  - Databases, computational biology, graph algorithms, etc.
- To get efficient sort, all computing resources need to be used
Why sorting?

• Sorting used as a important primitive in many applications
  – Databases, computational biology, graph algorithms, etc.

• To get efficient sort, all computing resources need to be used

Cannot just rely on clock rate

More DLP and TLP
Approaches to Data-Level Parallelism (i.e., SIMD)

- Compiler-based approaches
  - Compiler options
  -Pragma directives
Approaches to Data-Level Parallelism (i.e., SIMD)

- Compiler-based approaches
  - Compiler options
  -Pragma directives

**Issues:**
Fail to auto-vec loops, due to complex memory access, convoluted data rearrangement, etc.
Approaches to Data-Level Parallelism (i.e., SIMD)

• Compiler-based approaches
  – Compiler options
  –Pragma directives

• Manual optimization via ...
  – Compiler intrinsics
  – Assembly code
Approaches to Data-Level Parallelism (i.e., SIMD)

- Compiler-based approaches
  - Compiler options
  -Pragma directives
- Manual optimization via ...
  - Compiler intrinsics
  - Assembly code

*Issues:* Tedious and error-prone.
Approaches to Data-Level Parallelism (i.e., SIMD)

• Compiler-based approaches
  – Compiler options
  –Pragma directives

• Manual optimization via ...
  – Compiler intrinsics
  – Assembly code

Serial C codes

```c
for(i=0; i<2w; i++)
{
  if(i<w)
  else
    trgB[i-w] = i%2!=0 ? inpB[i/2] : inpA[i/2];
}
```

Ex. Interleave two arrays

AVX intrinsics on CPUs

```c
__mm256 v1 = _mm256_unpacklo_ps(inpA, inpB);
__mm256 v2 = _mm256_unpackhi_ps(inpA, inpB);
__mm256 trgA = _mm256_permute2f128_ps(v1, v2, 0x20);
__mm256 trgB = _mm256_permute2f128_ps(v1, v2, 0x31);
```

AVX512 intrinsics on MIC

```c
__mm512i l = _mm512_permute4f128_epi32(inpA, _MM_PERM_BDAC);
__mm512i h = _mm512_permute4f128_epi32(inpB, _MM_PERM_BDAC);
__mm512i t0 = _mm512_mask_swizzle_epi32(h, 0xcccc, l, _MM_SWIZ_REG_BADC);
__mm512i t1 = _mm512_mask_swizzle_epi32(l, 0x3333, h, _MM_SWIZ_REG_BADC);
__mm512i l = _mm512_mask_permute4f128_epi32(t1, 0x0f0f, t0, _MM_PERM_CDAB);
__mm512i h = _mm512_mask_permute4f128_epi32(t0, 0x0f0f, t1, _MM_PERM_CDAB);
__mm512i trgA = _mm512_shuffle_epi32(l, _MM_PERM_BDAC);
__mm512i trgB = _mm512_shuffle_epi32(h, _MM_PERM_BDAC);
```
Approaches to Data-Level Parallelism (i.e., SIMD)

• Compiler-based approaches
  – Compiler options
  –Pragma directives

• Manual optimization via ...
  – Compiler intrinsics
  – Assembly code

Serial C codes

```
for(i=0; i<2w; i++)
{
    if(i<w)
    else
        trgB[i-w] = i%2!=0 ? inpB[i/2] : inpA[i/2];
}
```

Ex. Interleave two arrays

```
__mm256 v1 = _mm256_unpacklo_ps(inpA, inpB);
__mm256 v2 = _mm256_unpackhi_ps(inpA, inpB);
__mm256 trgA = _mm256_permute2f128_ps(v1, v2, 0x20);
__mm256 trgB = _mm256_permute2f128_ps(v1, v2, 0x31);
```

AVX intrinsics on CPUs

```
__mm512i l = _mm512_permute4f128_epi32(inpA, _MM_PERM_BDAC);
__mm512i h = _mm512_permute4f128_epi32(inpB, _MM_PERM_BDAC);
__mm512i t0 = _mm512_mask_swizzle_epi32(h, 0xcccc, l, _MM_SWIZ_REG_BADC);
__mm512i t1 = _mm512_mask_swizzle_epi32(l, 0x3333, h, _MM_SWIZ_REG_BADC);
__mm512i l = _mm512_mask_permute4f128_epi32(t1, 0x0f0f, t0, _MM_PERM_CDAB);
__mm512i h = _mm512_mask_permute4f128_epi32(t0, 0xf0f0, t1, _MM_PERM_CDAB);
__mm512i trgA = _mm512_shuffle_epi32(l, _MM_PERM_BDAC);
__mm512i trgB = _mm512_shuffle_epi32(h, _MM_PERM_BDAC);
```

Can they be automatically generated?

Ex. Interleave two arrays
ASPaS Framework

- Formalize the data-reordering patterns in the parallel sorting
- Automatically generate the SIMD code
- Applied generally to DLP architecture, specific to x86 processors
Roadmap

- Introduction & Motivation
- Background
  - Sorting Networks & SIMD Processing
- ASPaS Framework
  - SIMD Sorter
  - SIMD Transposer
  - SIMD Merger
  - SIMD Code Generator
    ▶ Generate patterns for sorting the data segment by segment
    ▶ Generate patterns for merging the sorted data
    ▶ Generate codes from the patterns
- Evaluation & Discussion
- Conclusion
Background: Sorting Networks

- **Sorting Networks**
  - Comparisons can be planned out in a fixed pattern
  - Data flow is irrelevant with the value of input data
Background: SIMD for Intel Xeon Phi

- VPU Architecture
  - Manipulate one vector
Background: SIMD for Intel Xeon Phi

- VPU Architecture
  - Manipulate two vectors
Roadmap

• Introduction & Motivation
• Background
  – VPU Architecture & Sorting Networks
• ASPaS Framework
  – SIMD Sorter
  – SIMD Transposer
  – SIMD Merger
  – SIMD Code Generator

  ▶ Generate patterns for sorting the data segment by segment
  ▶ Generate patterns for merging the sorted data
  ▶ Generate codes from the patterns

• Evaluation & Discussion
• Conclusion
ASPaS Framework

• ASPaS Structure Overview

For users

Unsorted data → aspas_sort() → aspas_transpose() → Merge Stage → aspas_merge() → Sorted data

For developers

ASPaS Framework

- SIMD Code Generator
  - ISA-friendly primitive pool
  - Real SIMD inst.
  - translate

- Patterns
  - SIMD Sorter
  - SIMD Transposer
  - SIMD Merger

- Sorting networks
- Merging networks

Check validation
ASPaS Framework

- SIMD Sorter
  - Generate regrouped comparison patterns
  - Accept any kinds of sorting networks
ASPaS Framework

- SIMD Transposer
  - Why need the transpose?
ASPaS Framework

• SIMD Transposer
  – Why need the transpose?
ASPaS Framework

- SIMD Transposer
  - Generalize the patterns required in the in-register transpose

\[ \begin{array}{c}
\text{v0} \\
\text{v1} \\
\text{v2} \\
\text{v3}
\end{array} = \begin{array}{cccc}
3 & 1 & 5 & 2 \\
5 & 2 & 6 & 4 \\
7 & 5 & 8 & 5 \\
9 & 6 & 9 & 7 \\
\end{array} \]

\[ \begin{array}{cccc}
L_2^8 & I_2 \otimes L_2^4 & I_4 \otimes L_2^2 & L_4^8 \\
\end{array} \]

\[ \begin{array}{cccc}
v0_0 = 3 \\
v0_1 = 1 \\
v0_2 = 5 \\
v0_3 = 2 \\
v1_0 = 5 \\
v1_1 = 2 \\
v1_2 = 6 \\
v1_3 = 4 \\
v2_0 = 7 \\
v2_1 = 9 \\
v2_2 = 8 \\
v2_3 = 9 \\
v3_0 = 3 \\
v3_1 = 5 \\
v3_2 = 6 \\
v3_3 = 9 \\
\end{array} \]

Same pattern applies on v2 and v3

Same pattern applies on v1 and v3
ASPaS Framework

- SIMD Merger
  - Generalize the patterns required in the in-register merge

Inconsistent patterns
ASPaS Framework

• SIMD Merger
  – Generalize the patterns required in the in-register merge

\[
\begin{align*}
v_0 &= 3 \\
v_1 &= 5 \\
v_2 &= 7 \\
v_3 &= 9 \\
u_0 &= 6 \\
u_1 &= 5 \\
u_2 &= 2 \\
u_3 &= 1 \\
S_2 &\quad S_2 &\quad S_2 \\
S_2 &\quad S_2 &\quad S_2 \\
S_2 &\quad S_2 &\quad S_2 \\
\end{align*}
\]
ASPaS Framework

• SIMD Code Generator
  – Generate SIMD codes based on the received patterns
  – Primitive Pool Building

① Permuted Primitives  <Unique and symmetric data-reordering>

One vector
ASPaS Framework

• SIMD Code Generator
  – Generate SIMD codes based on the received patterns
  – Primitive Pool Building

① Permute Primitives  <Unique and symmetric data-reordering>

Suppose there are 4 units in vector

\[ 4^4 = 256 \text{ possible permutations} \]

Permutations w/o repetition => 4! = 24

Symmetric permutations => 8 DCBA(original order), DBCA, CDAB, BDAC, BADC, CADB, ACBD, and ABCD
ASPaS Framework

• SIMD Code Generator
  – Generate SIMD codes based on the received patterns
  – Primitive Pool Building

① Permute Primitives
② Blend Primitives  <Symmetric and equal data-blending>
ASPaS Framework

• SIMD Code Generator
  – Generate SIMD codes based on the received patterns
  – Primitive Pool Building

① Permute Primitives
② Blend Primitives  <Symmetric and equal data-blending>

Suppose there are 4 elements in vector

Only need 2 blend primitives to select every 1, 2 (1 to log(W)) elements from two input vectors respectively
• E.g. 1010

Part 1

Part 2

Kaixi@VT
8/10/2017
ASPaS Framework

• SIMD Code Generator
  – Sequence Building Algorithm
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm
• SIMD Code Generation
  – Sequence Building Algorithm

Initial Lane Check

Default Vector

Sequence Building Algorithm

Target Vector
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm

Diagram:
- Initial Lane
- Intra-lane Permute Primitive
- Blend Primitive
- Inter-lane Permute Primitive
- Elem Check
- Lane Check
- Default Vector
- Target Vector
- Sequence Building Algorithm
- BADC FEHG
- ABCD EFGH
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm

Diagram:
- Intra-lane Permute Primitive
- Initial Lane Check
- Blend Primitive
- Inter-lane Permute Primitive
- Elem Check
- Lane Check
- Selected Primitive
- Sequence Building Algorithm
- Target Vector
- Default Vector

Sequence Building Algorithm
ASPaS Framework

- SIMD Code Generator
  - Sequence Building Algorithm

Initial Lane Check

Intra-lane Permute Primitive

Elem Check

Default Vector

Sequence Building Algorithm

Target Vector

Blend Primitive

Inter-lane Permute Primitive

Selected Primitives

Selected Primitive Seq

ABCD EFGH

ABCD EFGH
ASPaS Framework

• SIMD Code Generator
  - Translate: selected primitive sequence to real codes
    • Intra-lane permute primitive => _mm512_shuffle
    • Inter-lane permute primitive => _mm512_permute4f128
    • Blend primitive => mask integrated to bond shuffle/permute instructions
  - Towards TLP
    • Threads sort their own parts (aspas::sort())
    • Half of them merge the adjacent parts (aspas::merge())
      - Continues until only one thread left
### Experiment Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC</td>
<td>Intel Xeon Phi 5110P</td>
</tr>
<tr>
<td>Code Name</td>
<td>Knights Corner</td>
</tr>
<tr>
<td># of Cores</td>
<td>60</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>1.05 GHz</td>
</tr>
<tr>
<td>L1/L2 Cache</td>
<td>32 KB/512 KB</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB GDDR5</td>
</tr>
<tr>
<td>Compiler</td>
<td>icpc 13.0.1</td>
</tr>
<tr>
<td>Compiler Options</td>
<td>-mmic -O3</td>
</tr>
<tr>
<td>Random Number Range</td>
<td>[0, DATA_SIZE]</td>
</tr>
</tbody>
</table>
Evaluation & Discussion

- Performance of Different Sorting Networks

  - ASPaS_sort(): More comparators, worse performance
  - ASPaS_merge(): “Consistent” variant consists of the SIMD-unfriendly interleaving data-reordering
Evaluation & Discussion

- **Vectorization Efficiency**

  - **Sort stage**: ASPaS still can outperform the auto-vec version, thanks to its contiguous memory access.
  - **Merge stage**: the complex data dependency prevents the compiler from auto-vectorizing the loops.

  ![Graph of Sort stage and Merge stage execution time normalized to ASPaS.](image)
Evaluation & Discussion

• Comparison to Sorting from Libraries

[Graphs showing performance comparison between different sorting algorithms]

- ASPaS sort outperforms other sorting tools from widely-used libraries

Lower the better
Discussion

• Portability
  – Easily ported to other x-86 multi-core CPU architectures
  – Only need to change the part of “Translate: primitives to real codes” in the SIMD Code Generator
    • Permute primitives => _mm256_shuffle/permute2f128
    • Blend primitives => e.g. _mm256_unpacklo/unpackhi
Conclusion

• ASPaS: a framework for the Automatic SIMDization of Parallel Sorting code generation
  – Formalizes the data-reordering operations
  – Fast and efficiently build the real instruction sequences
  – Can be applied to CPU as well

• Various parallel sorting codes generated with ASPaS
  – Significant vectorization efficiency
  – Can outperform tools from STL, Boost, and Intel TBB

THANK YOU!

More info: http://synergy.cs.vt.edu