Origami: A High-Performance Mergesort Framework

Arif Arman and Dmitri Loguinov
Texas A&M University
Agenda

» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
Motivation

» Mergesort is highly appealing in real-world sorting tasks for several reasons

• Distribution insensitive

MSB Radixsort
Poor unless uniform

Quicksort
Samplesort
Combsort
Certain worst-case inputs
Mergesort is highly appealing in real-world sorting tasks for several reasons:

- Sequential processing of input/output
Mergesort is highly appealing in real-world sorting tasks for several reasons:

- Well-suited for multi-core parallelization
- Yields new optimized kernels for small inputs
Motivation

» Many mergesort variants have been proposed, however ...

• None examine how to optimize individual phases of the sort pipeline

• Majority single threaded or, if parallel, bottlenecks on memory bandwidth

• Do not offer a unifying solution simultaneously optimized for scalar, SSE, AVX2 and AVX-512 architectures
Contribution

» Introduce Origami, a highly optimized, distribution-insensitive, parallel mergesort framework

» Formalize a four-phase computational model
  • Examine how to achieve maximum speed at each phase

» Develop end-to-end sort by efficiently connecting the optimized components

» Generalize the algorithms for Scalar, SSE, AVX2 and AVX-512

» Fastest mergesort (1.5-2x speedup) with near perfect scaling
» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
**Pipeline Overview**

Unsorted input, broken to L2 cache size blocks

- **P₁. Tiny sorters**

Binary merge

- **P₂. In-cache merge**

Sorted C size blocks

- **P₃. Out-of-cache merge**

Sorted N/k size lists

- **P₄. Out-of-cache merge w/ partitioning**

Final sorted output

Partition
Agenda

» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
In practice, presort every $m$ items with a different algorithm.

Sorting networks have proven to be the fastest option for such small sorts.

```c
swap(x, y):
    tmp = min(x, y)
    y = max(x, y)
    x = tmp
```

SIMD (single-instruction multiple-data) allows $W$ (SIMD_WIDTH) scalar swaps with a pair of `_mm_min`, `_mm_max` intrinsics.
Tiny Sorters: Outline

Load keys

\(W\) \(
\)

Sort columns

Transpose

Store sorted runs of length \(W\)

Sort every \(W\) keys in-register

Prior works

Typically, \(r = \# \) of registers \(R\)

Load keys

\(r\) \(
\)

Sort columns

Matrix-column merge

Transpose

Matrix-row merge

Store sorted run of length \(rW\)

Sort every \(rW\) keys in-register

Origami
Matrix-Column Merge (mcmerge)

Goal: sort matrix in column-major order
- Use merge networks (reduced from sorting networks)
- Group items of matrix in partial columns of $r/2 \times 1$
- Run swaps of corresponding merge network

With $\text{len(keygroup)} > 1$, replace min/max for a swap with MergeNetwork/container -- term this cswap

Drawback: With growing depth of merge network, shuffles become costlier for large $c$
Matrix-Row Merge (mrmerge)

(a) reverse bottom rows
(b) cswap
(c) sort rows

\[ \text{largest}(\text{row}_j) \leq \text{smallest}(\text{row}_{j+1}) \]

> Not significantly affected by increasing complexity of merge networks -- excellent for large matrix sizes

> However, has non-negligible minimum cost (e.g., two transposes)
  - Makes it inefficient for short sequences -- in contrast to mcmerge
Agenda

» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
Advancing Pointers

```c

load registers r₀, ..., r_{k-1} from A; A += kW
load registers r_k, ..., r_{2k-1} from B; B += kW

while A != endA and B != endB:
    rswaps for MergeNetwork2k
    store r₀, ..., r_{k-1} to C; C += kW
    reload r₀, ..., r_{k-1} from A or B
    move A or B forward by kW

merge keys left in registers and the unfinished list
```

» Present works mostly use branching comparisons
  • bmerge_v0

```c
if (A[0] < B[0]):
    reload from A; A += kW
else:
    reload from B; B += kW
```

» Some attempts at branchless but still room for improvement

» Origami provides the fastest, purely branchless solution
Advancing Pointers

load registers $r_0, \ldots, r_{k-1}$ from A; A += $kW$
load registers $r_k, \ldots, r_{2k-1}$ from B; B += $kW$
loadFrom = A; opposite = B;
while loadFrom != endA and loadFrom != endB:
    rswap for MergeNetwork2k
    store $r_0, \ldots, r_{k-1}$ to C; C += $kW$
    flag = loadFrom[0] < opposite[0]
    tmp = flag ? loadFrom : opposite
    opposite = flag ? opposite : loadFrom
    loadFrom = tmp
load $r_0, \ldots, r_{k-1}$ from loadFrom
loadFrom += $kW$

merge keys left in registers and the unfinished list

» Solution: bmerge_v3
  • Use two pointers: loadFrom, opposite
  • Update pointers based on flag
  • Always use loadFrom for next group of keys and end-of-buffer checks

» Up to 86% faster than v0

» Removes speculation from control flow and makes it distribution insensitive

» Additional boost with multiple simultaneous merges
Agenda

» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
Independent Merge ($P_3$)

» $P_2$ finishes when threads are done sorting lists of L2-cache-size $C$

» In $P_3$
  • Threads continue independent merges, but out-of-cache
  • Maximum achievable speed is that of `memcpy`
    • Skylake-X i7 CPUs with DDR4-3200 quad channel memory max: 37 GB/s
    • Vectorized `bmerge_v3` exhausts this with just 3 threads
    • One thread may be enough for older CPUs and dual channel memory

» Majority of existing works ignore and continue with binary merges
  • A few use desired $k$-way merges but with limitations
    • L3 residing shared merge tree with circular queue internal buffers ...
    • L2 residing dedicated tree with fixed buffer, fixed $k$, and encoding-decoding keys with insertion sort tie-breaker ...
Origami comes with L2-cache residing k-way merge trees (mtree)

Each node performs 4-way merge
• Binary merges internally
• Tiny intermediate buffers (64-128 B)
• Root and leaves remain large

k can be tuned
• Optimal choice depends on number of threads running, memory bandwidth, and L2 cache size
Cooperative Merge ($P_4$)

» Origami $P_4$ avoids bottleneck on memory bandwidth
  • Merge must utilize $\geq k$ sequences
  
  • $k$ selected optimally by $mtree$ in $P_3$

» Avoid stragglers by creating many small jobs
  • Reduce wait time for the fastest thread

  • Leader thread performs initial partition

  • All threads parallely partition further

  • Add $k$-way merge jobs to shared queue
    • Threads draw their workload in parallel
Agenda

» Introduction

» Pipeline Overview

» Tiny Sorters

» In-cache Merge

» Out-of-cache Merge

» Experiments
Setup

8-core Intel i7-7820X (Skylake-X)
L2 cache: 1 MB
Clock: 4.7 GHz (fixed)
SIMD Support: SSE, AVX2, AVX-512

16-core dual socket Intel Xeon E5-2690
L2 cache: 256 KB
Clock: 3.3 GHz
SIMD Support: SSE, AVX

32 GB DDR4-3200
Quad-channel

256 GB DDR3-1333
Quad-channel
## Table 2: Merge speed (B keys/s) in a $32 \times W$ matrix

<table>
<thead>
<tr>
<th>$B$</th>
<th>$K$</th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\chi$</td>
<td>mc</td>
<td>mr</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>10.39</td>
<td>3.75</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>6.26</td>
<td>3.52</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2.81</td>
<td>3.24</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.58</td>
<td>2.83</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>3.51</td>
<td>1.96</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.45</td>
<td>1.71</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.06</td>
<td>1.41</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>64+64</td>
<td>8</td>
<td>1.44</td>
<td>1.06</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>--</td>
<td>--</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
### Chunked-sort (Out-of-cache)

#### Table 11: Chunked speed in $C_3$ (M/s); $N = 256M$, $B = 32$

<table>
<thead>
<tr>
<th>chunk size $c$</th>
<th>SSE</th>
<th>AVX2</th>
<th>AVX-512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128 K</td>
<td>63</td>
<td>53</td>
<td>40</td>
</tr>
<tr>
<td>256 K</td>
<td>61</td>
<td>47</td>
<td>33</td>
</tr>
<tr>
<td>512 K</td>
<td>59</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td>1 M</td>
<td>57</td>
<td>41</td>
<td>28</td>
</tr>
<tr>
<td>2 M</td>
<td>55</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>4 M</td>
<td>54</td>
<td>37</td>
<td>23</td>
</tr>
<tr>
<td>8 M</td>
<td>52</td>
<td>35</td>
<td>21</td>
</tr>
<tr>
<td>16 M</td>
<td>50</td>
<td>33</td>
<td>20</td>
</tr>
<tr>
<td>32 M</td>
<td>48</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>64 M</td>
<td>47</td>
<td>30</td>
<td>18</td>
</tr>
<tr>
<td>128 M</td>
<td>45</td>
<td>29</td>
<td>17</td>
</tr>
<tr>
<td>256 M</td>
<td>44</td>
<td>28</td>
<td>17</td>
</tr>
</tbody>
</table>

SSE: 110%  
AVX2: 100%  
AVX-512: 53%  

Define Checkpoint $C_i$ = execution of phases $P_1$ through $P_i$.
Distribution Insensitivity

Scalar

SSE

AVX2

AVX-512

D1: Uniform
D2: All same
D3: Sorted
D4: Reverse sorted
D5: Almost sorted (7th = MAX)
D6: Pareto
D7: Bursts of same keys (length from D6, key from D1)
D8: Random shuffle of D7
D9: Fibonacci
**Multi-core Speedup**

**Scalar**

**AVX2**

**SSE**

**AVX-512**

1 GB
Multi-core Speedup (Xeons)

SSE

64 GB
Database Queries (Xeons)

» IRLbot query

```sql
SELECT dst, COUNT(*) as cnt
FROM A INNER JOIN B ON A.src=B.src
WHERE A.outdeg < 1000000
GROUP BY dst
ORDER BY cnt DESC
```

» TPC-H queries

» Scaling factor: 100

Single-core: 37-60x
All-cores: 30-113x
Concluding Remarks

» Origami offers a highly optimized mergesort framework
  • Runs in a fast, constant speed for different data distributions
  • Gains a nearly linear speed-up in multi-core environments

» The proposed components are flexible to accommodate future SIMD extension sets
  • Programmer only needs to write a few arch-specific intrinsics

» Future work will examine
  • External memory sorting
  • Longer key/value pairs
  • Incorporation into existing DBMS
Thank You

- Arif Arman
- arman@tamu.edu
- https://arif-arman.github.io