Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort

Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. NGuyen, Victor W. Lee, Daehyun Kim, Pradeep Dubey – SIGMOD’10

Presentation by Stian Liknes
Abstract

• Comparison of sorting algorithms on modern architectures
  – Radix sort vs. Merge sort
  – Intel Core i7 vs. NVIDIA GTX 280
  – Best performing implementations on both architectures

• Performance analysis terms of architectural features
  – Core count, SIMD, and bandwidth

• Simulate 64-core platform ("sneak-peek into future")
  – Varying SIMD widths under constant bandwidth
Introduction 1

- Rapid increase in main memory
  - Main memory sorting feasible
  - Bounded by compute, bandwidth and latency characteristics of processor architecture

- Increase in compute capacity
  - More cores (thread-level parallelism)
  - Wide vector units (SIMD – data level parallelism)

- Memory bandwidth increasing at a slower pace
  - Bandwidth bound algorithms will not scale well in the future
Introduction 2

- Performance characteristics (algorithms)
  - Computational complexity
  - Architectural friendliness
  - Trade-off between factors

- Radix-sort
  - Low computational complexity $O(n)$
  - Not naturally data-parallel
  - Many passes over each data item leads to high bandwidth utilization

- Merge-sort
  - Moderate computational complexity $O(n \log n)$
  - Data-parallel merging networks
  - Bandwidth friendly
Introduction 2

• Performance characteristics (algorithms)
  – Computational complexity
  – Architectural friendliness
  – Trade-off between factors

• Radix-sort
  – Low computational complexity $O(n)$
  – Not naturally data-parallel
  – Many passes over each data item leads to high bandwidth utilization

• Merge-sort
  – Moderate computational complexity $O(n \log n)$
  – Data-parallel merging networks
  – Bandwidth friendly
Introduction 3

• Architecture-dependent implementations
  – Different bottlenecks such as irregular memory-access, lack of SIMD use, and conflicts in local storage.
  – CPU Radix sort uses a buffer in cache to localize scatters
    • Avoids capacity and conflict misses
  – GPU Radix sort uses 1-bit split based code
    • SIMD friendly
Related work

• Bitonic sort
  – SIMD friendly Merge-sort
  – Higher algorithmic complexity of $O(N \log^2 N)$
  – Well suited for GPUs before advent of scatter and local stores

• AA-Sort
  – Local comb-sort
  – Merge results

• Merge sort by Chhugani

• CC-Radix sort
  – Cache conscious
  – Does not account for cache conflict misses
Parallelism

• **Thread-level parallelism**
  - Combining or splitting different blocks of data in parallel
  - Blocking input data and using global histogram updates to coordinate between blocks (radix-sort)

• **Data-level parallelism**
  - Harder to exploit on certain algorithms
  - Need data laid out contiguously in memory to avoid gather/scatter
  - Radix sort involves irregular memory accesses and common histogram updates making it hard to optimize for SIMD
  - Merge sort can use sorting networks
Memory bandwidth

- Sorting is typically memory intensive
- Should avoid bandwidth dependent algorithms
- Architectures bridge gap with on-chip local storage
  - Cache hierarchies on CPUs
  - Shared memory on GPUs
  - If data fits, no main memory bandwidth is utilized
  - Merge-sort modified to use only two reads and two writes of data
    - Multi-way merge that limits working set to fit in caches
Latency Effects/ILP

• Caches and TLBs have minimal misses on streaming access
  – Caches organized into cache lines of 64 bytes
    • Consecutive accesses belong to the same cache line => cache hit
  – Cache conflict misses
    • Different memory regions mapped to the same cache line
    • Alternately accessed (avoided if streaming)
• Merge-sort has streaming access pattern
• Radix-sort has a data rearrangement step where contiguous inputs are written into widely scattered output locations
  – Buffer scheme to minimize cache conflict misses
Radix-sort (RS)

• Basic algorithm
  – Break input into digits and sort a digit at a time
  – Use counting sort to sort keys according to a given digit

• Basic parallel algorithm (thread-level parallelism)
  1) Divide input evenly among T threads and compute a local histogram Ht for each thread t
  2) Compute a global histogram from the local histograms using a parallel prefix sum operation
  3) Each thread computes the write offset for each key in its partition and key is scattered to correct position
Architecture friendly RS

• Bottlenecks
  – Irregular memory accesses in computing histograms
  – Rearranging of a big array in step 3 (bandwidth-unfriendly)

• Improve locality of scatters by utilizing local storage
  – Two methods:
    1) Buffer up writes to different cache line wide regions of main memory in local stores and writing them in a contiguous manner (bandwidth utilization, fewer page misses)
    2) Locally sorting small blocks of data that fit into local memory according to the considered radix (SIMD friendly) – sort one bit at a time in each block
RS Implementation on CPUs

- Specifications (quad-core Core i7)
  - 4-wide SSE (SIMD)
  - 2 SMT threads per core to hide latency
  - 32 KB L1, 256KB L2 shared per core

- Split radix sort bad performance
  - Low SIMD width => Not worth the extra instructions

- Buffer implementation most efficient
  - Absence of gather/scatter SSE instructions
  - Histogram update and buffer scatter implemented in scalar code
  - Best buffer- and radix size depends on cache size and number of TLB entries ($B*K =$ multiple of 64 bytes)
  - Buffer and histograms must fit in the 128 KB on-die L2 cache per thread

Compute bound for current CPUs
RS Implementation on GPUs

• Specifications (NVIDIA GTX 280)
  – 30 shared multiprocessors (SMs)
  – Up to 32 threads per SM
  – 8 scalar processors per SM (8-wide SIMD)
  – 32-wide logical SIMD (= 1 thread warp)

• Split radix good performance
  – Utilize wide SIMD
  – Data divided evenly into a set of thread blocks
    • Locally sorted by 1-bit stream splits.
    • Each thread computes a histogram of given bins based on local sort
    • Global prefix used to calculate placements

• Buffer implementation bad performance
  – No SIMD utilization (only 1 of 32 operations per warp)
Merge-sort (MS)

• Basic algorithm
  – Merge two sorted lists of size L into a sorted list of size 2L, merge 2L to 4L, and so on until there is only one sorted list left

• Scalar version suffers from branch mispredictions

• Data-parallel variant uses merging networks
  – Infrequent branches
  – Bitonic merge network
  – Successively merge lists at SIMD width granularity
  – Lists to be merged distributed among threads
  – Bandwidth-oblivious by adopting multi-way merge
MS Implementation on CPUs

- Using multi-way merge data only read and written once from/to main memory (bitonic)
- SIMD-friendly and bandwidth-oblivious

```
// The following code is run three times for a 4x4 network
L_1 = sse_min(A,B);  // A and B are the two input SSE registers
H_1 = sse_max(A,B);
L_1p = sse_shuffle(L_1, H_1, imm2);
H_1p = sse_shuffle(L_1, H_1, imm3);
```
MS Implementation on GPUs

- Similar to CPU using bitonic network
  - 16 wide sorting network
  - Absence of single-instruction shuffle operations, need to implement manually by scattering elements
  - Scattering to shared memory creates bank conflicts
    - Avoided by permuting the lists to be merged
  - Not bandwidth-bound, even if bandwidth decreases with 4x

/* A single level of the merge network.
read and write offsets are permuted to avoid bank conflicts */
__device__ void BitonicMergeLevel(uint * buf){
  a = buf[readoff_1];
  b = buf[readoff_2];
  buf[writeoff_1] = min(a, b);
  buf[writeoff_2] = max(a, b);
  __syncthreads();
}

/* buf is present in __shared__ space.
buf[0..63] initially contains the two input arrays with the second half reversed */
__device__ void BitonicMergeNetwork(uint * buf){
  for(uint i = 0; i < 5; i++){
    // Each iteration represents one level
    // of the bitonic merge network
    BitonicMergeLevel(buf);
  }
}
Performance evaluation 1

• Radix sort
  – Best performance on CPU
  – GPU use split-code that has many more instructions than the buffer code (needed to utilize 32-wide SIMD)

• Merge-Sort
  – Best performance on GPU, but small difference
  – GPU lacks a single instruction scatter
Performance evaluation 2

![Graphs showing sorting rate vs. log(input list size) for CPU Radix with different bit sizes.](image-url)
Performance evaluation 3

CPU Merge

GPU Merge
## Comparison to other sorts

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW</td>
<td>30.0</td>
<td>141.7</td>
</tr>
<tr>
<td>GFlops</td>
<td>103.0</td>
<td>933.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>256K</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
<td>2.1</td>
<td>2.1</td>
<td>1.3</td>
<td>2.5</td>
</tr>
<tr>
<td>1M</td>
<td>4.4</td>
<td>5.2</td>
<td>5.3</td>
<td>9.6</td>
<td>5.4</td>
<td>5.0</td>
<td>9.1</td>
</tr>
<tr>
<td>4M</td>
<td>17.2</td>
<td>25.4</td>
<td>23.3</td>
<td>40.9</td>
<td>20.8</td>
<td>21.6</td>
<td>38.1</td>
</tr>
<tr>
<td>16M</td>
<td>67.6</td>
<td>160.7</td>
<td>101.5</td>
<td>185.7</td>
<td>81.7</td>
<td>94.5</td>
<td>139.8</td>
</tr>
<tr>
<td>64M</td>
<td>271.0</td>
<td>550.5</td>
<td>439.7</td>
<td>835.5</td>
<td>333.4</td>
<td>381.8</td>
<td>524.3</td>
</tr>
</tbody>
</table>
Large keys and payloads 1

- Fixed key length keys
  - Maximize use of SIMD
  - Minimize use of bandwidth
  - Remap variable length keys to fixed length keys of 4 or 8 bytes

- Impact on Radix Sort
  - Increase in compute and bandwidth resources used
  - Number of passes doubles as key lengths double
  - Bandwidth requirement scales quadratically with key length
  - CPU can utilize unused SSE registers to store keys up to 16 bytes, making instructions per pass constant
  - GPU already use SIMD registers => increase in instructions
Large keys and payloads 2

• Impact on Merge Sort
  – Bandwidth requirement increases linearly with key length
  – Compute requirements depends on
    • Data types
    • Number of keys compared in a single SIMD instruction
    • Gain in performance by using smaller merge networks
      – If SIMD is used to store the entire key
  – CPU
    • 64 bit SIMD comparison efficient
    • Lack of gather support
  – GPU
    • No support for 64 bit SIMD comparison
    • Efficient gathering
Large keys and payloads 3

Graph showing relative sorting time for different key sizes (32-bit, 64-bit, 128-bit) and payloads for Radix Sort and Merge Sort on CPU and GPU.
Future architectures

• Predictions
  – Increase in core count
  – Increase in SIMD width
  – Bandwidth unlikely to scale with increased core count

• Effect of increasing SIMD
  – Radix will rely on SIMD-friendly split version
  – Merge sort 6.5X increase on 16X wider SIMD
    • Wider merge networks and overheads that are not SIMD friendly
  – Radix sort unable to utilize SIMD resources efficiently
    • Bandwidth restrictions

• Effect of increasing memory size
  – Bigger databases in memory
  – Bigger keys
Future architectures 2

![Graph showing SIMD scalability vs. SIMD width for different architectures: Merge, Radix-32, Radix-64, Radix-128. The graph plots SIMD scalability on the y-axis against SIMD width (8wide, 16wide, 32wide, 64wide, 128wide) on the x-axis. The scalability increases with SIMD width for all architectures.](image-url)
Conclusions

- Radix sort faster on current architectures
- Gap narrows from CPUs to GPUs
- Merge sort performs better when sorting large keys
- Radix sort predicted to hit a “bandwidth wall” as SIMD width increases
- Merge sort predicted to utilize increased SIMD with efficiently