High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs

Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and Paolo Ienne

Key Laboratory of Computer System and Architecture
ICT, CAS, China
Outline

- GPU computation model
- Our sorting algorithm
  - A new bitonic-based merge sort, named Warpsort
- Experiment results
- Conclusion
GPU computation model

- Massively multi-threaded, data-parallel many-core architecture
- Important features:
  - SIMT execution model
    - Avoid branch divergence
  - Warp-based scheduling
    - implicit hardware synchronization among threads within a warp
  - Access pattern
    - Coalesced vs. non-coalesced
Why merge sort?

- Similar case with external sorting
  - Limited shared memory on chip vs. limited main memory
- Sequential memory access
  - Easy to meet coalesced requirement
Why bitonic-based merge sort?

- Massively fine-grained parallelism
  - Because of the relatively high complexity, bitonic network is not good at sorting large arrays
  - Only used to sort small subsequences in our implementation
- Again, coalesced memory access requirement
Problems in bitonic network

- naïve implementation
  - Block-based bitonic network
  - One element per thread
- Some problems
  - in each stage
    - \( n \) elements produce only \( n/2 \) compare-and-swap operations
    - Form both ascending pairs and descending pairs
  - Between stages
    - synchronization

Too many branch divergences and synchronization operations
What we use?

- Warp-based bitonic network
  - each bitonic network is assigned to an independent warp, instead of a block
    - Barrier-free, avoid synchronization between stages
  - threads in a warp perform 32 distinct compare-and-swap operations with the same order
    - Avoid branch divergences
    - At least 128 elements per warp

- And further a complete comparison-based sorting algorithm: GPU-Warpsort
Overview of GPU-Warpsort

Divide input seq into small tiles, and each followed by a warp-based bitonic sort.

Merge, until the parallelism is insufficient.

Split into small subsequences.

Merge, and form the output.
Step 1: barrier-free bitonic sort

- divide the input array into equal-sized tiles
- Each tile is sorted by a warp-based bitonic network
  o 128+ elements per tile to avoid branch divergence
  o No need for __syncthreads()
  o Ascending pairs + descending pairs
  o Use max() and min() to replace if-swap pairs

```c
bitonic_warp_128(key_t *keyin, key_t *keyout){
    for(i = 0; i < log128N; i++){
        for(j = i; j < 128; j += 2){
            if(keyin[j] > keyin[j + 1])
                swap(keyin[j], keyin[j + 1]);
            k = position of preceding element in each pair to form descending order
            if(keyin[k] > keyin[k + 1])
                swap(keyin[k], keyin[k + 1]);
        }
    }
}
```

---

ICT Institute of Computing Technology, Chinese Academy of Sciences
Step 2: bitonic-based merge sort

- $t$-element merge sort
  - Allocate a $t$-element buffer in shared memory
  - Load the $t/2$ smallest elements from seq $A$ and $B$, respectively
  - Merge
  - Output the lower $t/2$ elements
  - Load the next $t/2$ smallest elements from $A$ or $B
  - $t = 8$ in this example
Step 3: split into small tiles

● Problem of merge sort
  o the number of pairs decreases geometrically
  o Can not fit this massively parallel platform

● Method
  o Divide the large seqs into independent small tiles which satisfy:

    sequence \sqsubseteq \text{subsequence}\ y \text{jab}
Step 3: split into small tiles (cont.)

- How to get the splitters?
  - Sample the input sequence randomly

```
Input sequence: ...

Sample sequence: ...

Sorted sample sequence: ...

Splitters: ...
```
Step 4: final merge sort

- Subsequences $(0, i), (1, i), \ldots, (l-1, i)$ are merged into $S_i$
- Then, $S_0, S_1, \ldots, S_l$ are assembled into a totally sorted array
Experimental setup

● Host
  o AMD Opteron880 @ 2.4 GHz, 2GB RAM

● GPU
  o 9800GTX+, 512 MB

● Input sequence
  o Key-only and key-value configurations
    ■ 32-bit keys and values
  o Sequence size: from 1M to 16M elements
  o Distributions
    ■ Zero, Sorted, Uniform, Bucket, and Gaussian
Performance comparison

- Mergesort
  - Fastest comparison-based sorting algorithm on GPU (Satish, IPDPS’09)
  - Implementations already compared by Satish are not included

- Quicksort
  - Cederman, ESA’08

- Radixsort
  - Fastest sorting algorithm on GPU (Satish, IPDPS’09)

- Warpsort
  - Our implementation
Performance results

- **Key-only**
  - 70% higher performance than quicksort

- **Key-value**
  - 20%+ higher performance than mergesort
  - 30%+ for large sequences (>4M)
Results under different distributions

- Uniform, Bucket, and Gaussian distribution almost get the same performance
- Zero distribution is the fastest
- Not excel on Sorted distribution
  - Load imbalance
Conclusion

- We present an efficient comparison-based sorting algorithm for many-core GPUs
  - carefully map the tasks to GPU architecture
    - Use warp-based bitonic network to eliminate barriers
  - provide sufficient homogeneous parallel operations for each thread
    - avoid thread idling or thread divergence
  - totally coalesced global memory accesses when fetching and storing the sequence elements

- The results demonstrate up to 30% higher performance
  - Compared with previous optimized comparison-based algorithms
Thanks