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

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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

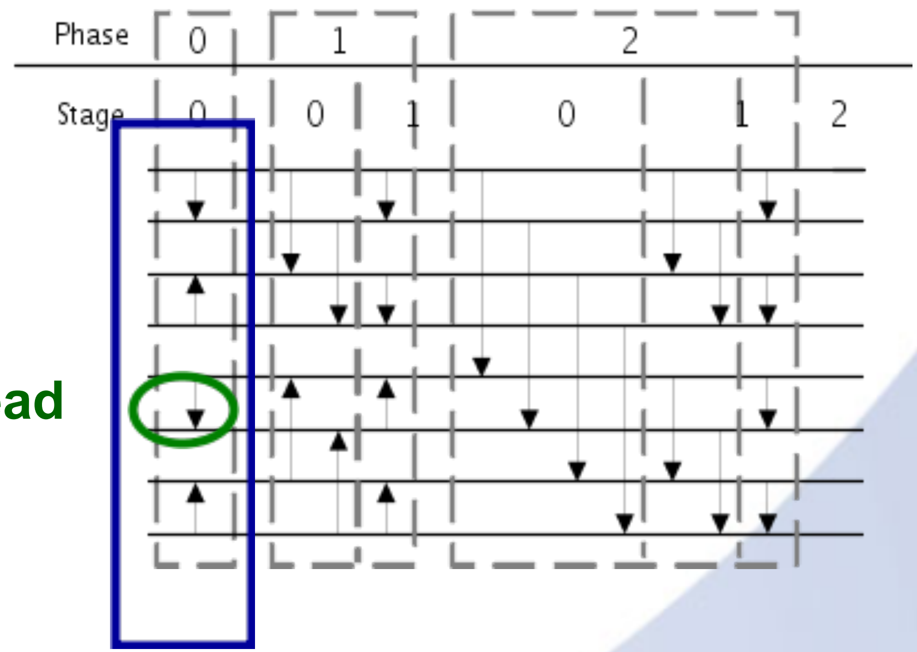
Why bitonic-based merge sort ?

- 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

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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

ICT 中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

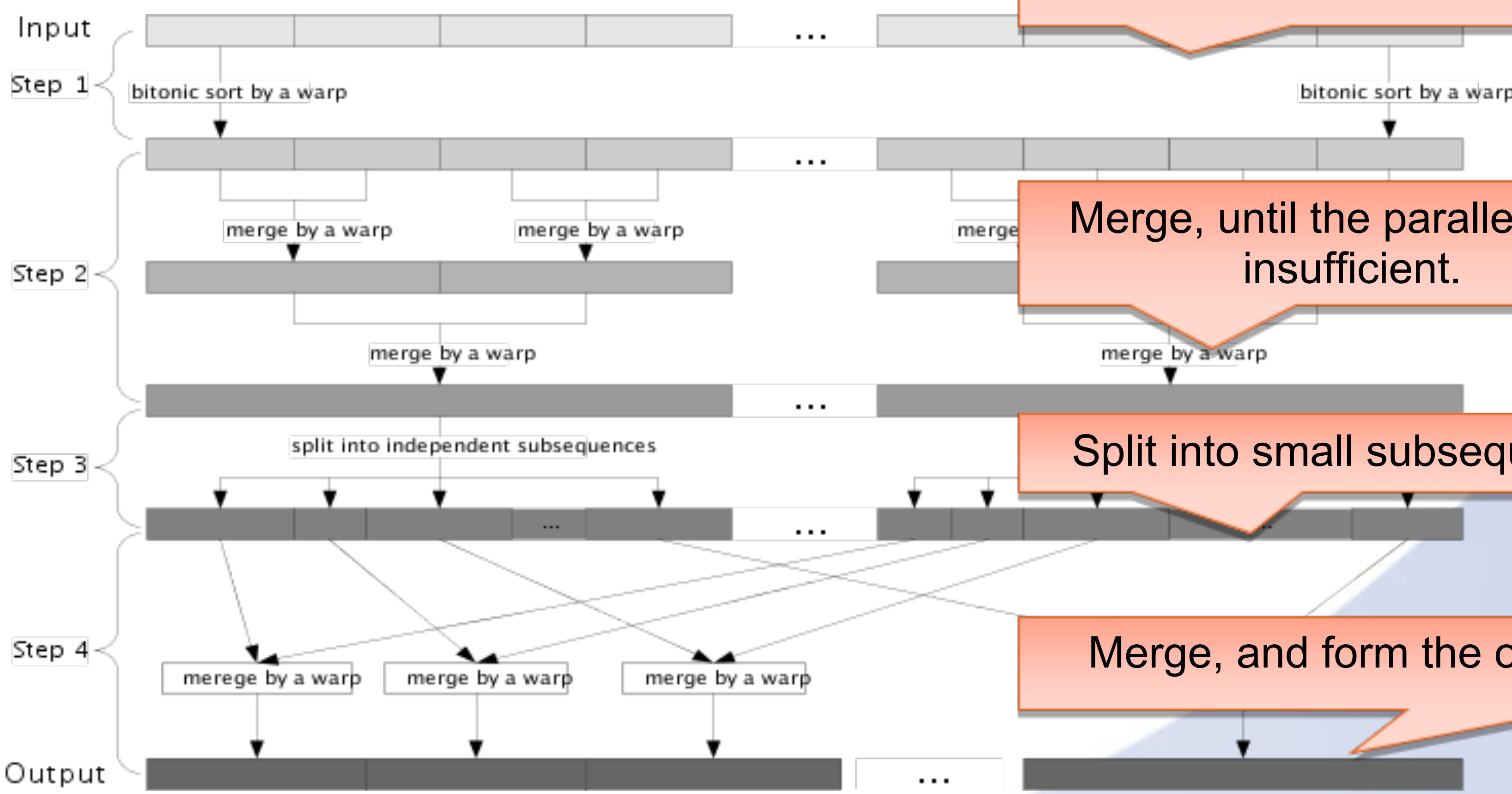
Overview of GPU-Warpsort

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



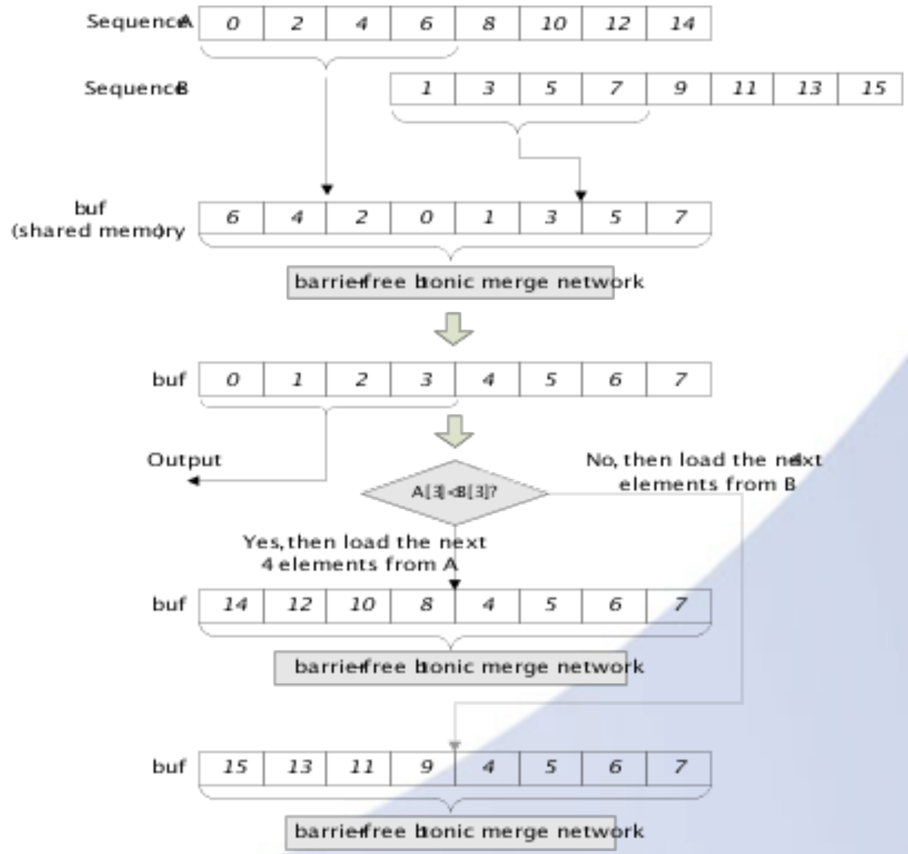
Step1: barrier-free bitonic sort

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

```
bitonic_warp_128_(key_t *keyin, key_t *keyout){
    //phase 0 to log(128)-1
    for(i=2;i<128;i*=2){
        for(j=i/2;j>0;j/=2){
            k0 ← position of preceding element in each pair
                to form ascending order
            if(keyin[k0]>keyin[k0+j])
                swap(keyin[k0],keyin[k0+j]);
            k1 ← position of preceding element in each pair
                to form descending order
            if(keyin[k1]<keyin[k1+j])
                swap(keyin[k1],keyin[k1+j]);
        }
    }
    //special case for the last phase
    for(j=128/2;j>0;j/=2){
        k0 ← position of preceding element in the thread's
            first pair to form ascending order
        if(keyin[k0]>keyin[k0+j])
            swap(keyin[k0],keyin[k0+j]);
        k1 ← position of preceding element in the thread's
            second pair to form ascending order
        if(keyin[k1]>keyin[k1+j])
            swap(keyin[k1],keyin[k1+j]);
    }
}
```

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



中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

Step 3: split into small tiles

- Problem of merge sort
  - the number of pairs decreases geometrically
  - Can not fit this massively parallel platform
- Method
  - Divide the large seqs into independent small tiles which satisfy:

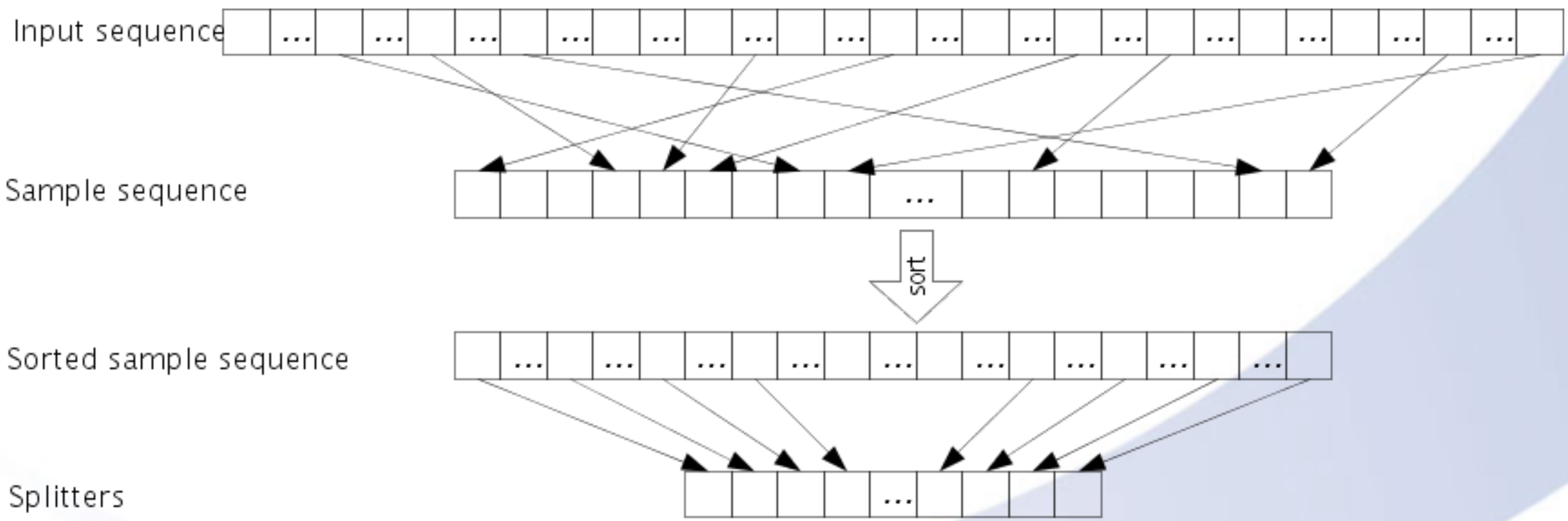
[illegible] *subsequencexibsubsequenceyjab*

[illegible] £ < <

INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

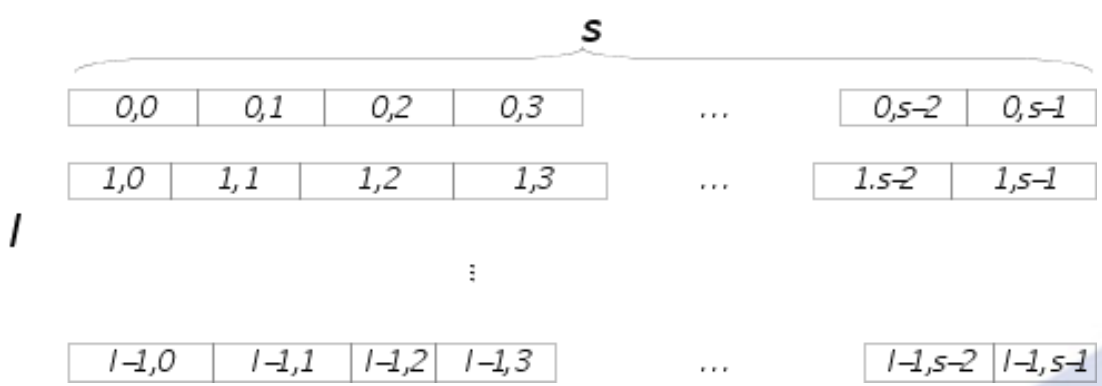
Step 3: split into small tiles (cont.)

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



Step 4: final merge sort

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



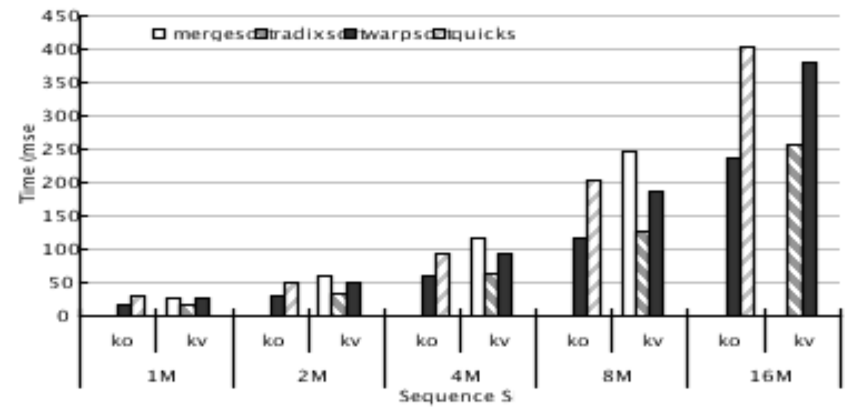
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

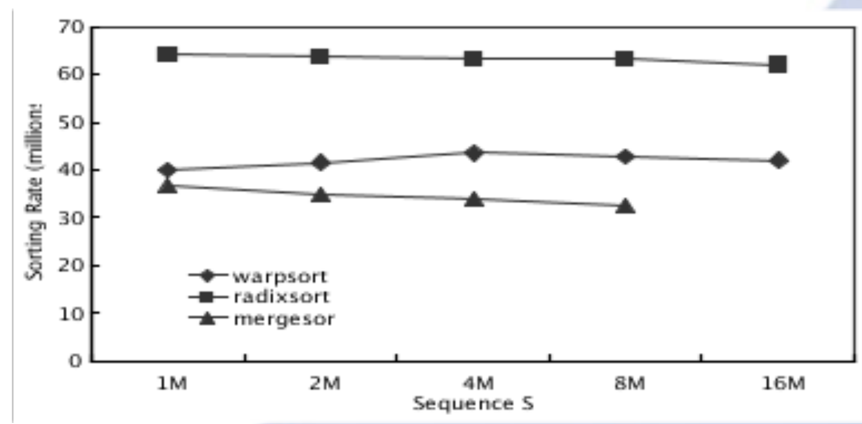
Experimental setup

- Host
  - AMD Opteron880 @ 2.4 GHz, 2GB RAM
- GPU
  - 9800GTX+, 512 MB
- Input sequence
  - Key-only and key-value configurations
    - 32-bit keys and values
  - Sequence size: from 1M to 16M elements
  - 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





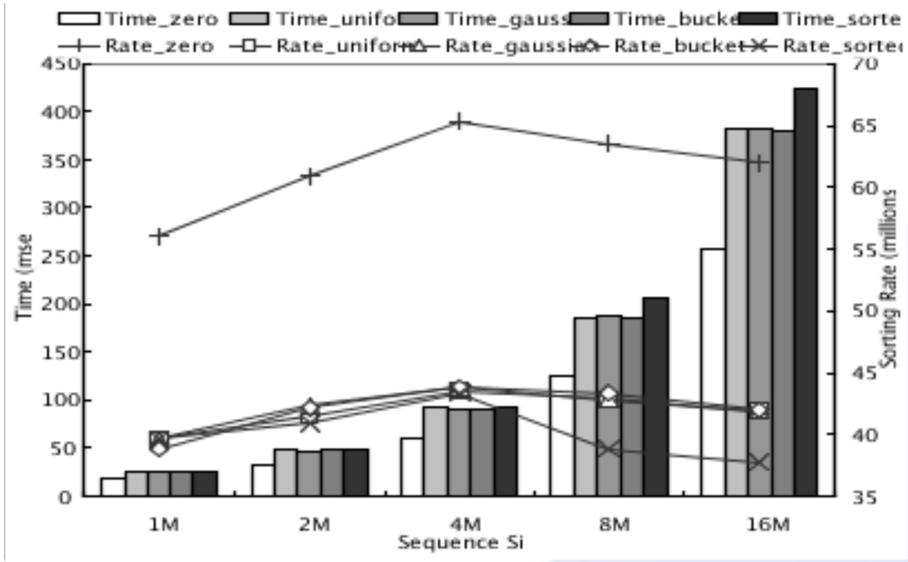
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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



中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

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