Parallel Sorting

Michael Garland

NVIDIA Research

Problem Overview

- Given a sequence of *n* integers, called keys

```
A = [8 4 3 9 0 9 7]
```

- Place keys in output in non-decreasing order

```
sorted(A) = [0 3 4 7 8 9 9]
```

- Optionally with equal values in their original order
  - "stable" sorts provide this; "unstable" sorts do not

Why Sorting?

- Put data in order
- Make searching easier
- Build data structures in parallel
- … and many others

Some assumptions for today

- Keys are integers of fixed length (e.g., 32 bits)
- Keys are not part of larger records
- Sequences reside entirely in main memory
- “Main memory” of the processor we’re using
  - in CPU memory for CPU sorts
  - in GPU memory for GPU sorts

Sorting problems we won't discuss

- External memory sorting
  - data doesn't fit in memory all at once
- Distributed sorting
  - data resides in physically separate memories
- Long and/or variable length keys
  - can significantly change performance trade offs
- Among others …

How do we sort?

Some simple sorts

- **Selection**
  - remove the smallest key of the input
  - append at the end of the output
  - repeat
- **Insertion**
  - remove the next key of the input
  - insert into the output in sorted order
  - repeat

Sequential (mostly)

- **Transposition**
  - find pair where `A[i]>A[i+1]` and swap them
  - repeat until there are none

Parallel (potentially)

Odd-Even Transposition Sort

- **Parallelizing transposition sort:**
  - **assign 1 thread to each element**
  - **use odd/even phases to prevent contention**

requires at most n/2 iterations

```
while A is not sorted:
    if is_odd(i) and (A[i+1] < A[i])
          swap(A[i], A[i+1])
    barrier
    if is_even(i) and (A[i+1] < A[i])
          swap(A[i], A[i+1])
    barrier
```

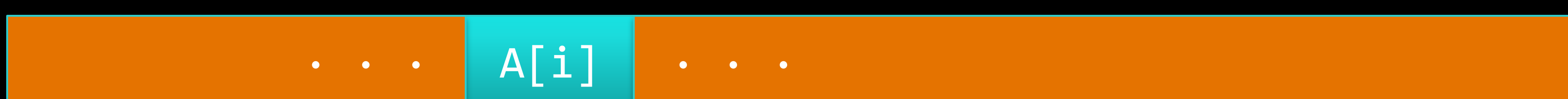
Counting Sort

- **Step 1: Count elements sorting to left of `A[i]`**

```
rank[i] = count( j<i where A[j] ≤ A[i] )
        + count( j>i where A[j] < A[i] )
```

- **Step 2: Scatter to position in sorted order**

```
permute( A[i] -> A[rank[i]] )
```



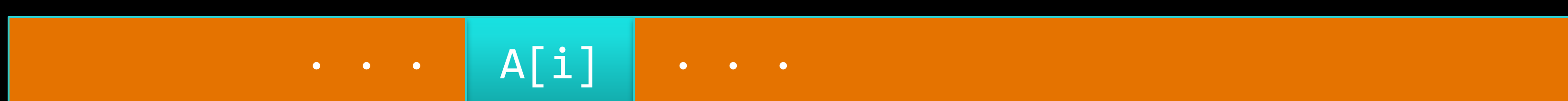
Counting Sort (alternate)

- **Step 1: Count places that `A[i]` needs to move**

```
offset[i] = count( j<i where A[j] > A[i] )
          - count( j>i where A[j] < A[i] )
```

- **Step 2: Scatter to position in sorted order**

```
permute( A[i] -> A[i-offset[i]] )
```



Binary Counting Sort

- **If `A[i]` is 0:**

```
offset[i] = count( j<i where A[j] == 1 )
```

- **If `A[i]` is 1:**

```
offset[i] = -count( j>i where A[j] == 0 )
```

- **And scatter:**

```
permute( A[i] -> A[i-offset[i]] )
```

A Simple Radix Sort

Apply binary counting sort to each bit of the keys, from LSB to MSB

```
def radix_sort(A, msb=32):
       def delta(flag, ones_before, zeros_after):
           if flag==0:  return -ones_before
           else:        return +zeros_after

       lsb = 0

       while lsb<msb:

           flags = [(x>>lsb)&1 for x in A]
           ones  = scan(plus, flags)
           zeros = rscan(plus, [f^1 for f in flags])

           offsets = map(delta, flags, ones, zeros)
           A = permute_with_offsets(A, offsets)

           lsb = lsb+1

       return A
```

Is this efficient?

Apply binary counting sort to each bit of the keys, from LSB to MSB

```
def radix_sort(A, msb=32):
       def delta(flag, ones_before, zeros_after):
           if flag==0:  return -ones_before
           else:        return +zeros_after

       lsb = 0

       while lsb<msb:

           flags = [(x>>lsb)&1 for x in A]
           ones  = scan(plus, flags)
           zeros = rscan(plus, [f^1 for f in flags])

           offsets = map(delta, flags, ones, zeros)
           A = permute_with_offsets(A, offsets)

           lsb = lsb+1

       return A
```

Radix Sort

- Apply counting sort to successive digits of keys
- Performs *d* scatter steps for *d*-digit keys
- Scattering in memory is fundamentally costly

Parallel Radix Sort

- **Assign tile of data to each block** **(1024 elements)**
- **Build per-block histograms of current digit** **(4 bit)**
- **Combine per-block histograms** **(P x 16)**
- **Scatter**

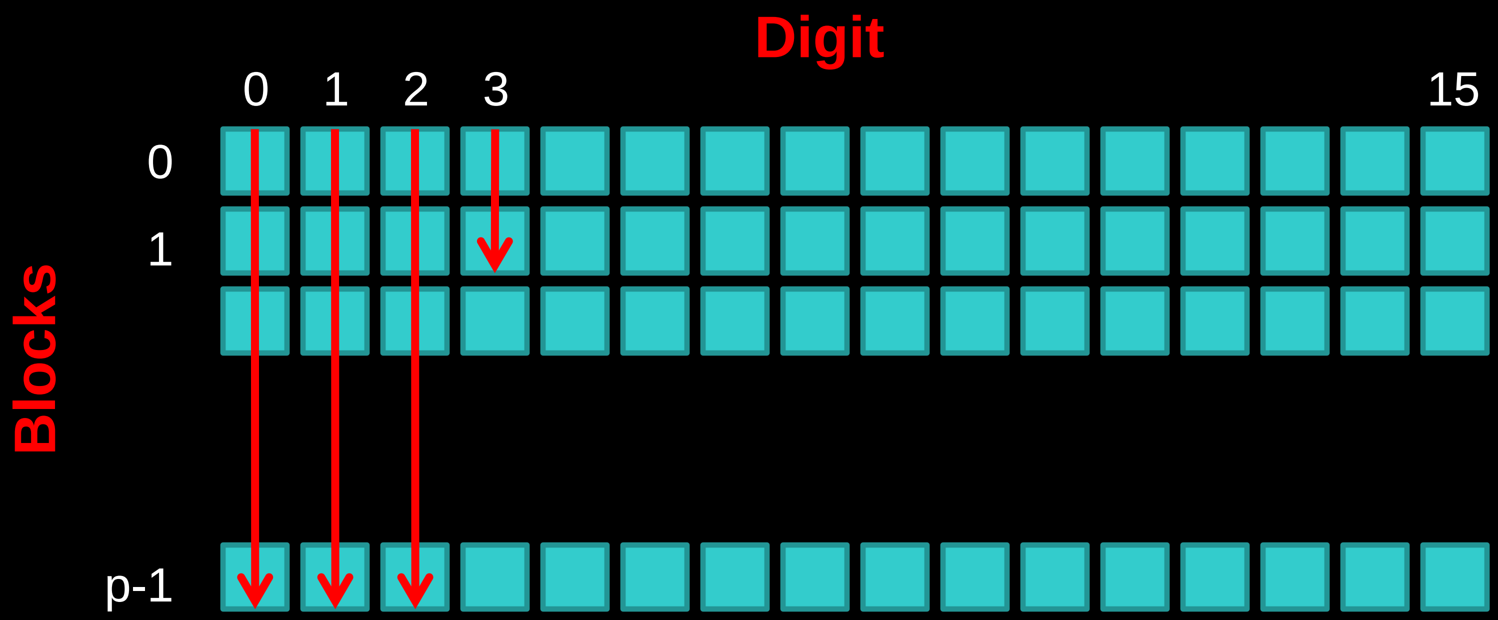
cf. Satish et al., *Designing efficient sorting algorithms for manycore GPUs*, IPDPS 2009.

Per-Block Histograms

- Perform b parallel splits for b-bit digit
- Each split is just a prefix sum of bits
  - each thread counts 1 bits to its left
- Write bucket counts & partially sorted tile
  - sorting tile improves scatter coherence later

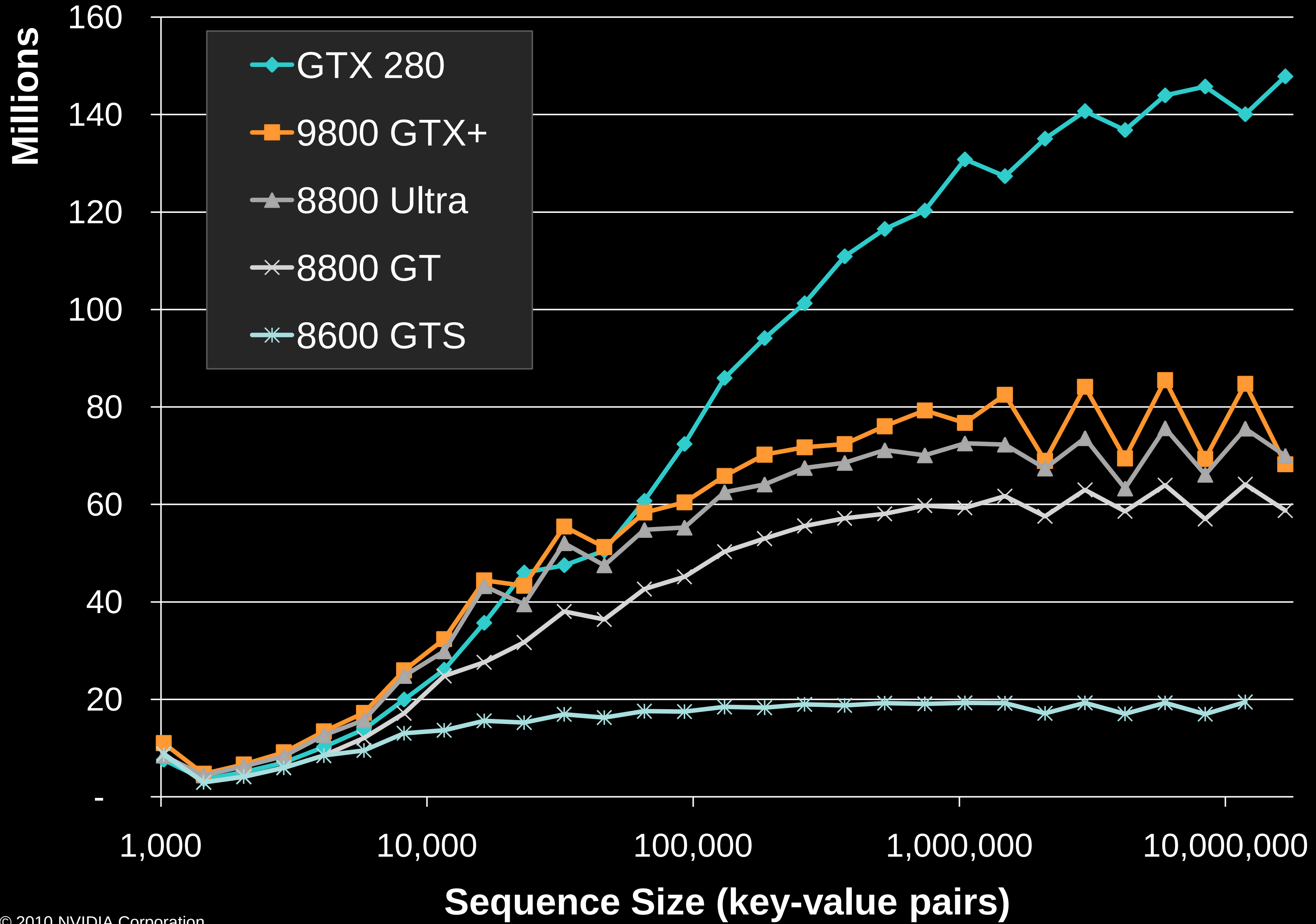
Combining Histograms

- **Write per-block counts in column major order & scan**



cf. Zagha & Blelloch, *Radix sort for vector multiprocessors*, SC'91.

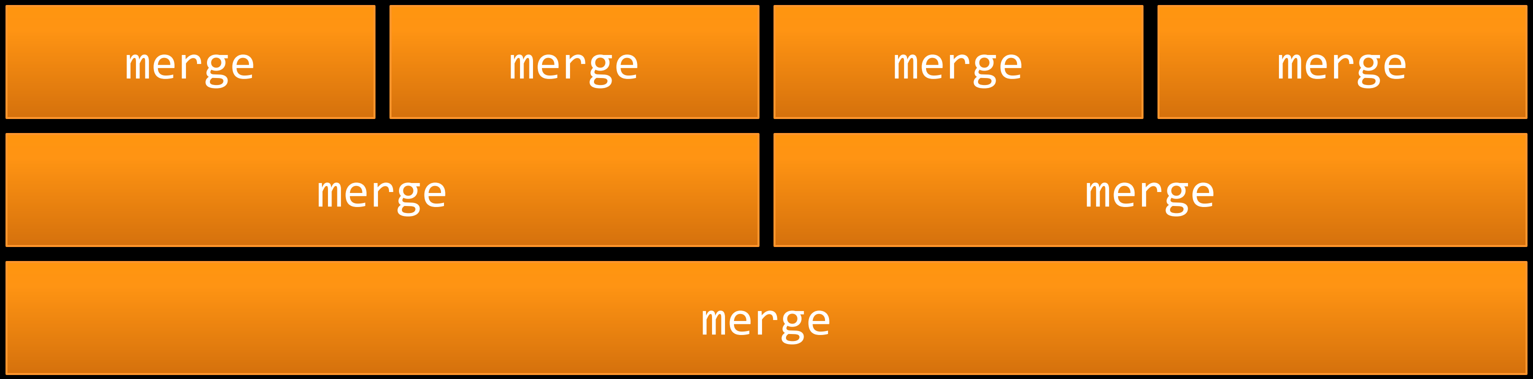
Radix Sorting Rate (pairs/sec)



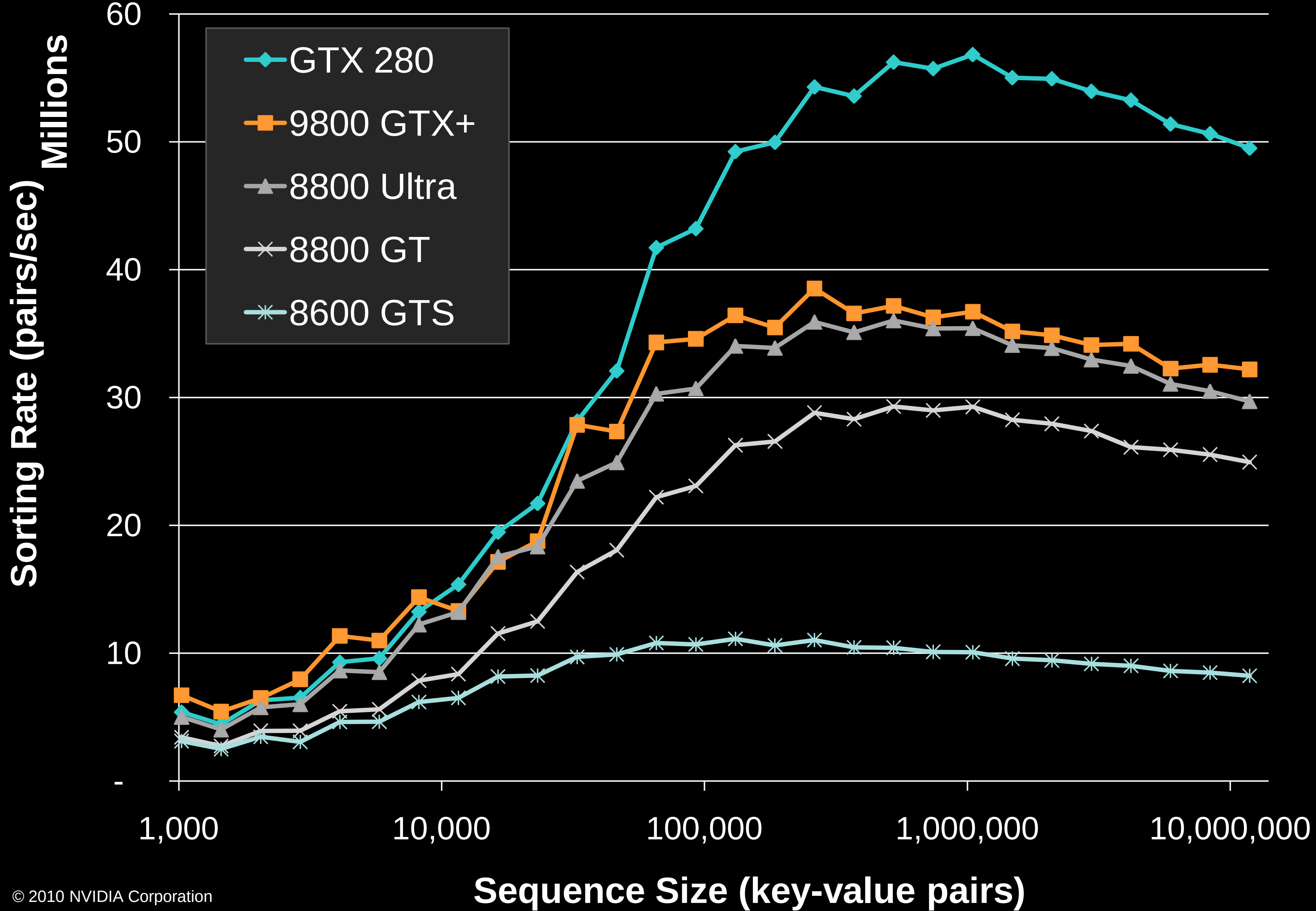
Merge Sort

- Divide input array into 256-element tiles
- Sort each tile independently

- Produce sorted output with tree of merges



Merge Sorting Rate



Some other techniques

- Quicksort / Sample Sort
  - partition keys into non-overlapping ranges
  - sort each range individually
- Sorting networks
  - fixed network of comparison operators
  - e.g., bitonic sort, odd-even merge sort

Questions?

mgarland@nvidia.com

Odd-Even Merge Sort

```
template<typename T, typename Cmp>
__device__ void oddeven_sort(T *keys, int i, int n, Cmp lt)
{
    for(unsigned int p=n/2; p>0; p/=2) {
        unsigned int q=n/2, r=0, d=p;
        while( q>=p ) {
            if( i<(n-d) && (i&p)==r ) {
                unsigned int j = i+d;
                T xi = keys[i], xj = keys[j];

                if( lt(xj,xi) ) {
                    keys[i] = xj;
                    keys[j] = xi;
                }
            }

            d = q-p; q = q/2; r = p;
            __syncthreads();
        }
    }
}
```

Algorithm M, Section 5.2.2
The Art of Computer Programming, Vol 3
D. E. Knuth