

First Swedish Workshop on Multi-Core
Computing MCC 2008 Ronneby:

**On Sorting and Load Balancing on
Graphics Processors**

Daniel Cederman and Philippas Tsigas
Distributed Computing Systems
Chalmers University of Technology

CUDA Seminar: Kristian Stavåker, krsta@ida.liu.se

Questions

- ▶ Name at least three sorting algorithms that have been implemented to be used on a GPU.
- ▶ Briefly describe the two *phases* of the GPU-Quicksort algorithm presented.
- ▶ Which two GPU NVIDIA hardware configurations were used for measurements?

Overview

- ❑ **Introduction**
- ❑ **CUDA Programming Model**
- ❑ **Sorting on GPUs**
- ❑ **Quicksort**
- ❑ **GPU-Quicksort**
- ❑ **Measurements**
- ❑ **(Load-balancing)**
- ❑ **Conclusions**

Introduction



Introduction

▶ CPU

- ▶ Multi-core
- ▶ Large cache
- ▶ Few threads
- ▶ Speculative execution

▶ GPU

- ▶ Many-core
- ▶ Small cache
- ▶ Wide and fast memory bus
- ▶ Thousands of threads hides memory latency
- ▶ Massive parallelism

Introduction (cont.)

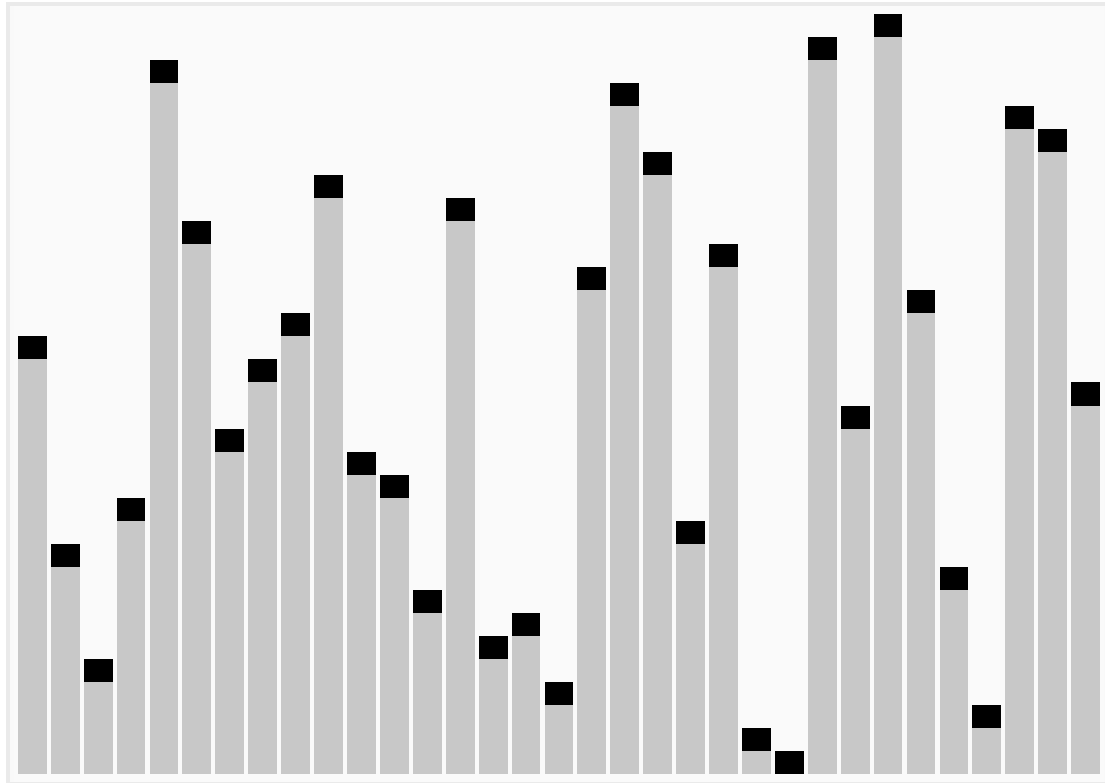
- ▶ CUDA – Compute Unified Device Architecture
- ▶ (Minimal) extensions to C/C++
- ▶ Designed for general purpose computation
 - ▶ Until recently the only way to take advantage of the GPU was to transform the problem into the graphics domain and use the tools available there.



Introduction (cont.)

- ▶ General sorting algorithms: Quicksort, Merge sort, Bucket sort, Insertion sort, Bubble sort, Heapsort, Radix sort, Shell sort, Selection sort, etc..
- ▶ In this presentation: GPU-Quicksort. We shall compare it with other GPU sorting implementations.
- ▶ Sequential Quicksort runs in $O(n \cdot \log(n))$ on average but in $O(n^2)$ in the worst case.

Introduction (cont.)



CUDA Programming Model

CUDA Software Stack

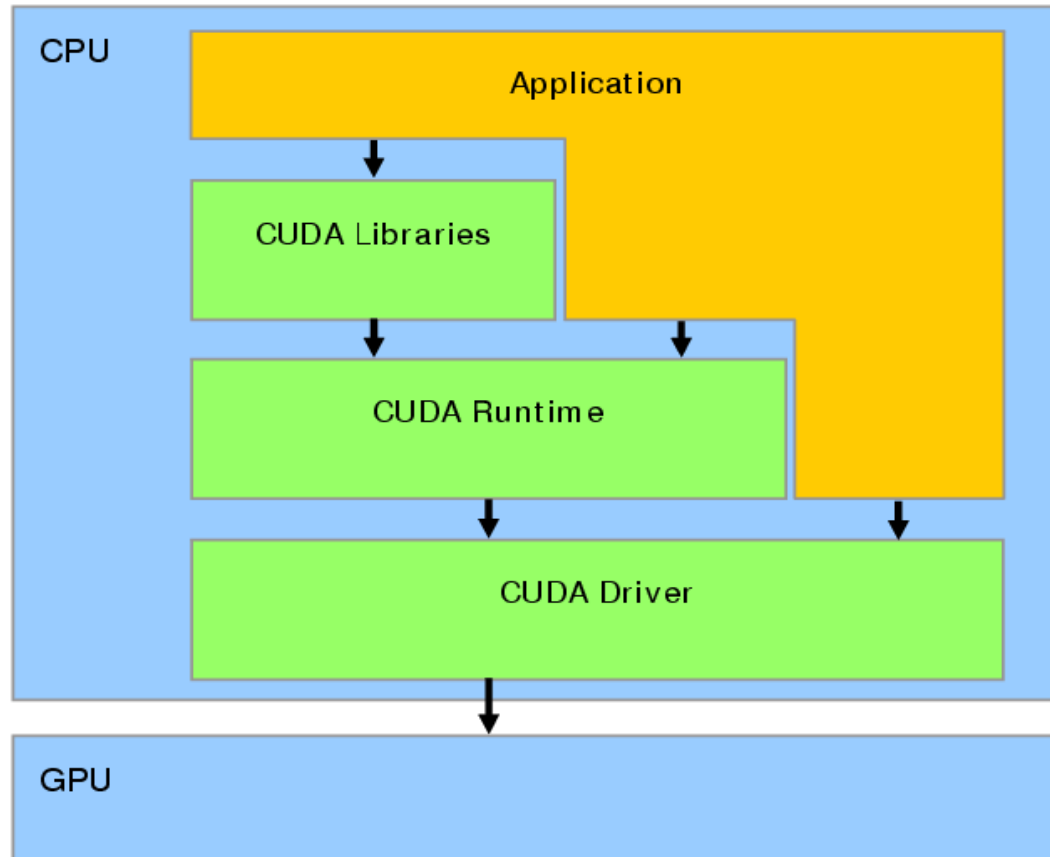


Figure 1-3. Compute Unified Device Architecture Software Stack

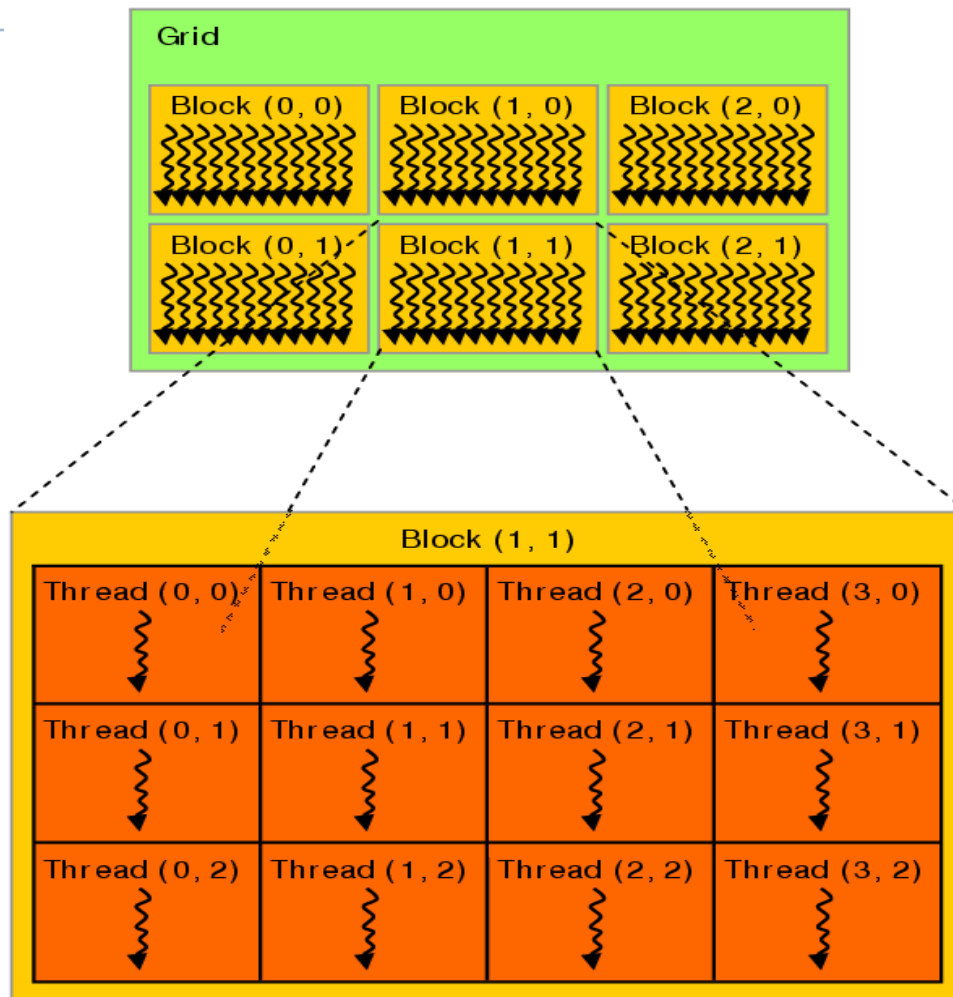
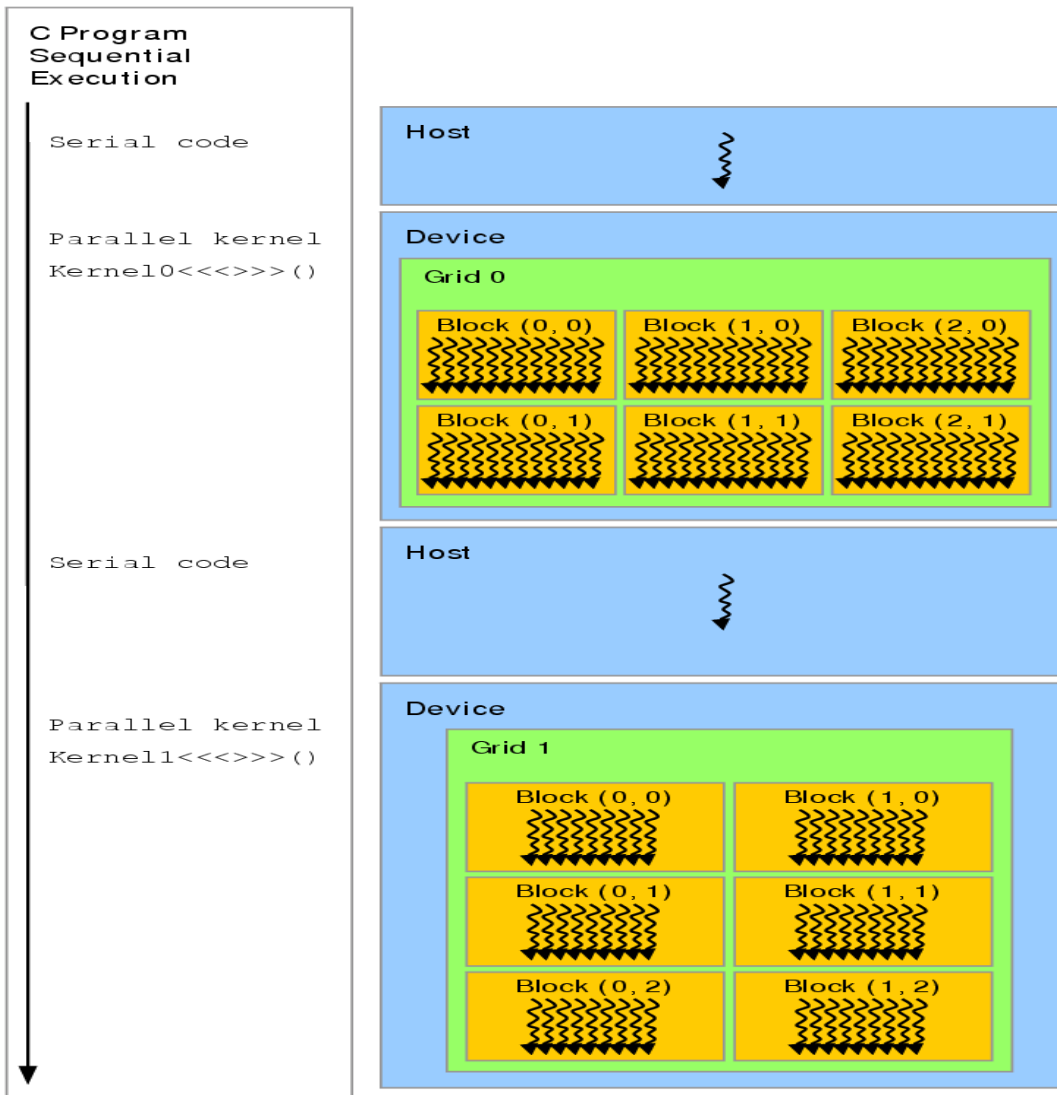
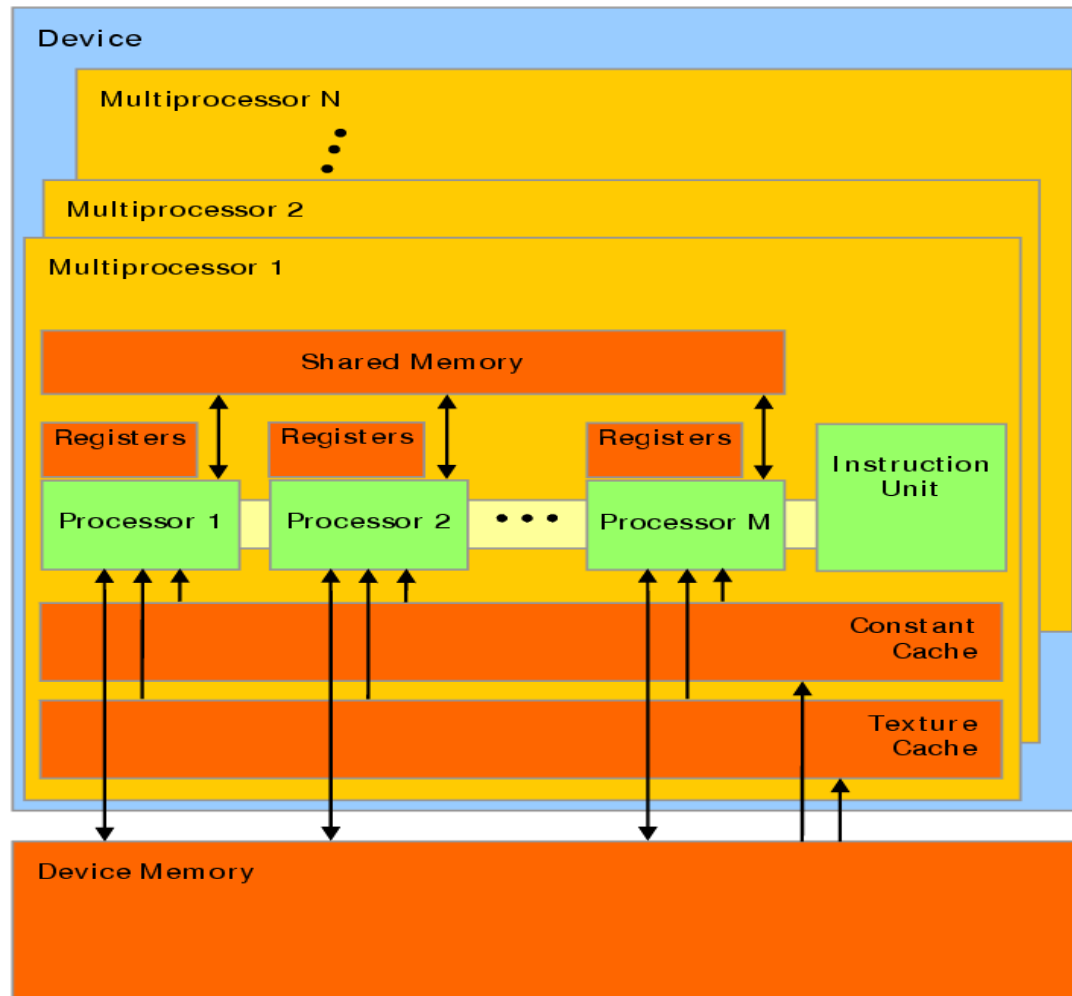


Figure 2-1. Grid of Thread Blocks



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming



A set of SIMT multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

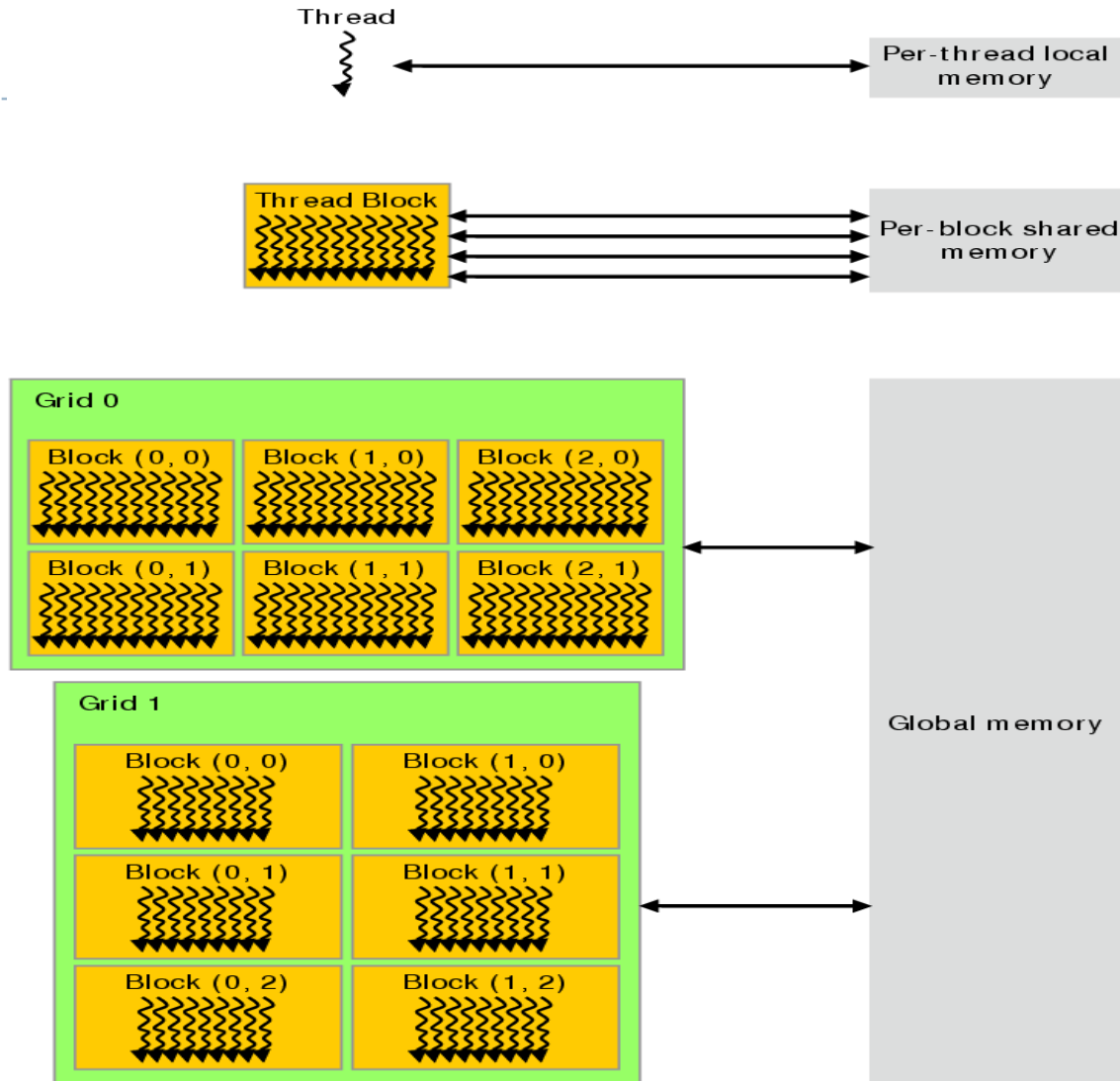


Figure 2-2. Memory Hierarchy

Sorting on GPUs

Sorting on GPUs

- ▶ GPU-Quicksort, Cederman and Tsigas, ESA08
- ▶ GPUSort, Govindaraju et. al., SIGMOD05
- ▶ Radix-Merge, Harris et. al., GPU Gems 3 '07
- ▶ Global radix, Sengupta et. al., GH07
- ▶ Hybrid, Sintorn and Assarsson, GPGPU07

CPU:

- ▶ Introsort, David Musser, Software: Practice and Experience

Quicksort

Quicksort

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

- Pick an element, called a pivot, from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted.

Quicksort Example

Pick a pivot element

23 | 12 9 2 7

Move elements

1 2 7 23 12 9

Subsequences, pick pivot elements

1 2 7 | 23 12 9

Move elements

1 2 7 9 23 12

Subsequences, pick pivot elements

1 | 2 7 9 | 23 12

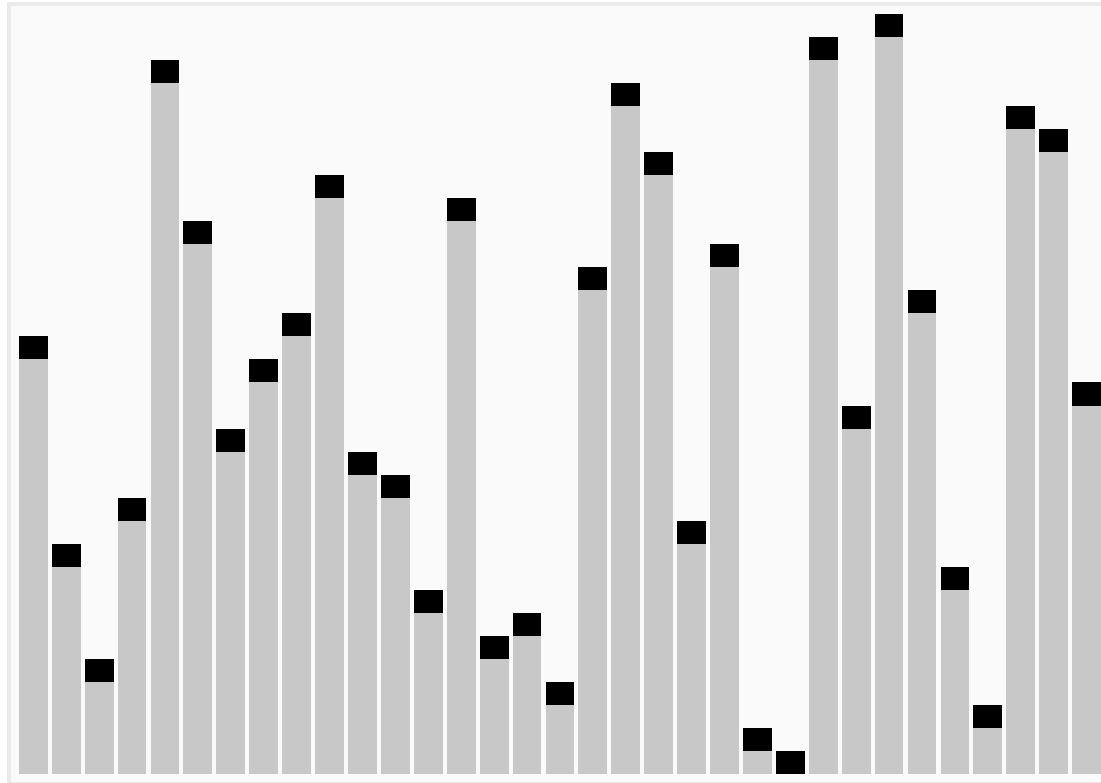
Move elements

1 | 2 7 9 | 12 23

Base case, done!

1 2 7 9 12 23

Quicksort Visualization



(Wikipedia)

Quicksort In-place

function partition(array, left, right, pivotIndex)

 pivotValue := array[pivotIndex]

 swap array[pivotIndex] and array[right] // Move pivot to end

 storeIndex := left

for i **from** left **to** right - 1

if array[i] ≤ pivotValue

 swap array[i] and array[storeIndex]

 storeIndex := storeIndex + 1

 swap array[storeIndex] and array[right] // Move pivot to its final place **return** storeIndex

procedure quicksort(array, left, right)

if right > left select a pivot index (e.g. pivotIndex := left)

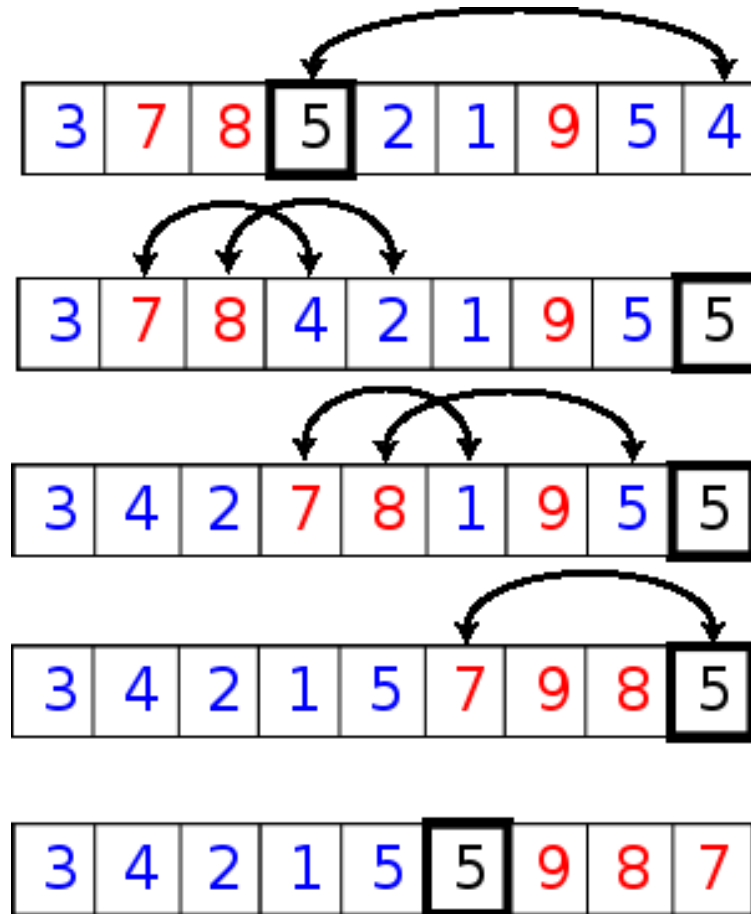
 pivotNewIndex := partition(array, left, right, pivotIndex)

 quicksort(array, left, pivotNewIndex - 1)

 quicksort(array, pivotNewIndex + 1, right)

(Wikipedia)

Quicksort In-place, Partitioning



(Wikipedia)

GPU-Quicksort

GPU-Quicksort

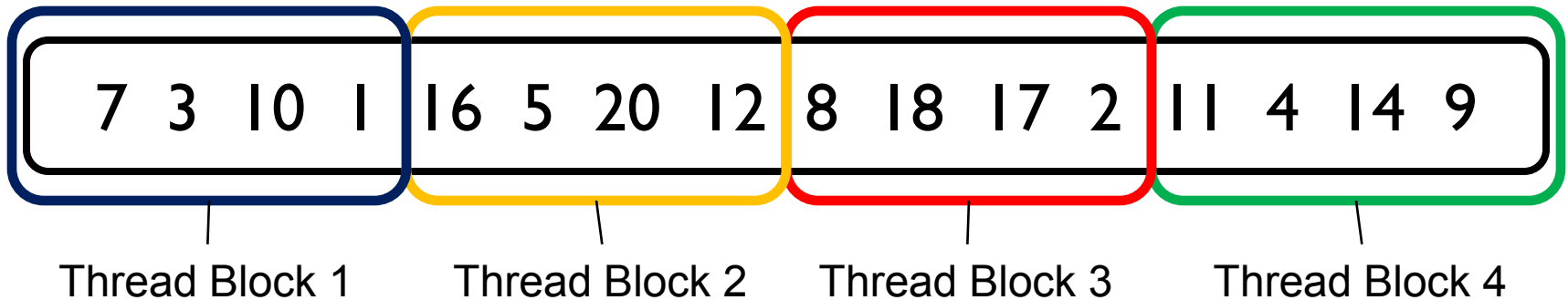
- ▶ Until now Quicksort has not been considered to be an efficient sorting algorithm for GPUs.
- ▶ This implementation achieves efficiency by using a two-phase design to keep thread synchronization low and by steering the threads so that their memory read operations are performed coalesced.
- ▶ We also try to take advantage of atomic primitive operations such as FAA - Fetch-And-Add (might not be available on all GPUs).

Outline

- ▶ Given some input data to be sorted (located in the GPU global memory) ...

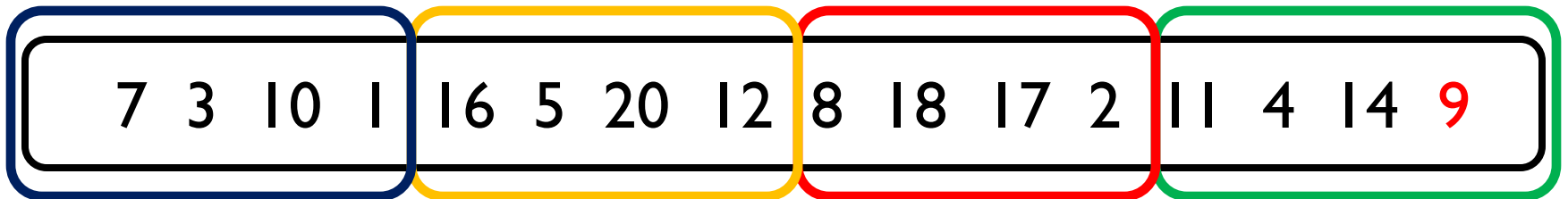
7 3 10 1 16 5 20 12 8 18 17 2 11 4 14 9

- ▶ First step: The sequence to be sorted is divided and thread blocks assigned to the subsequences ...



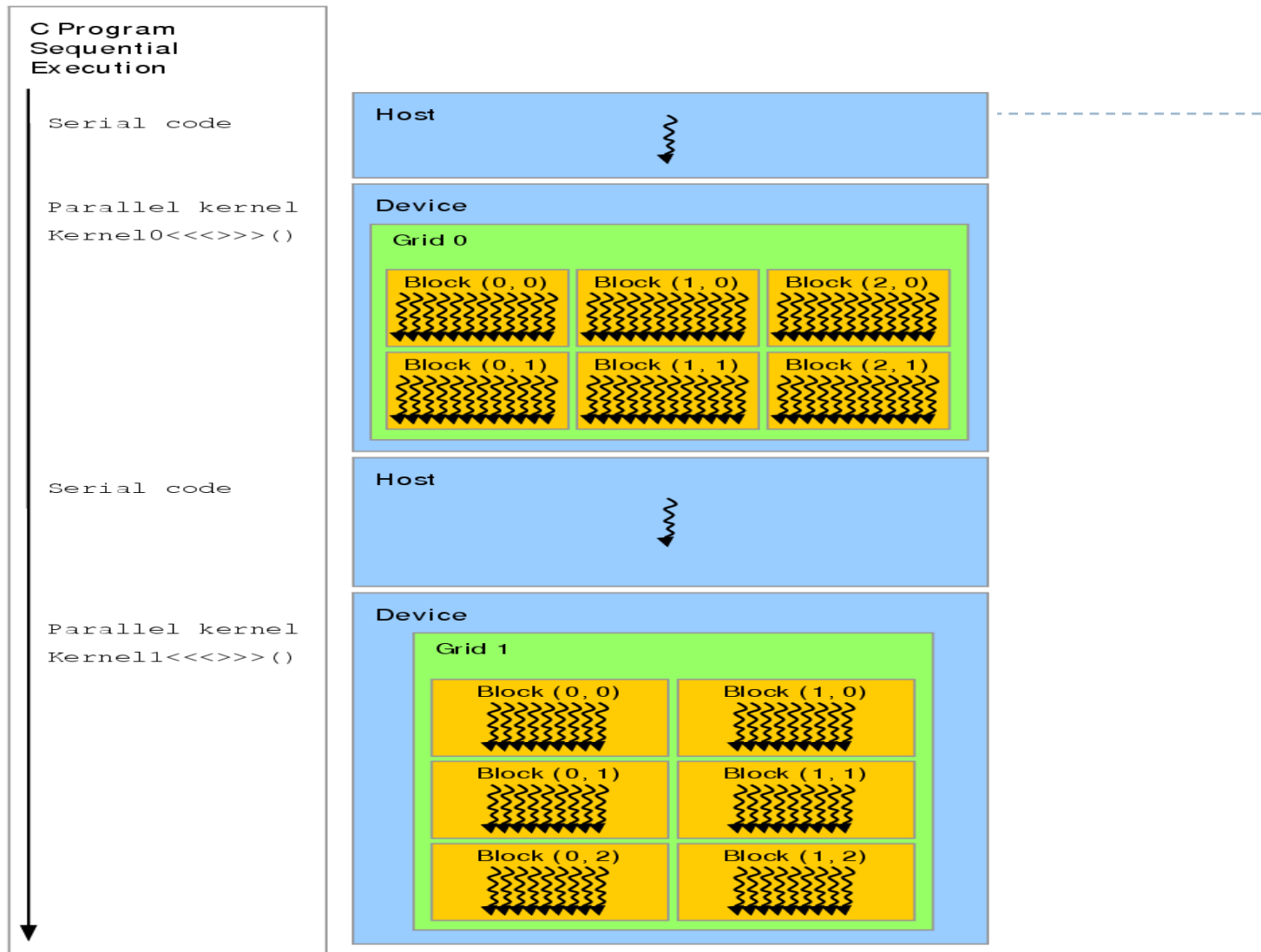
Outline (cont.)

- ▶ Pick a pivot element and partition the sequence into two. Thread block synchronization will be needed.



Partitioning ... More details later!





Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

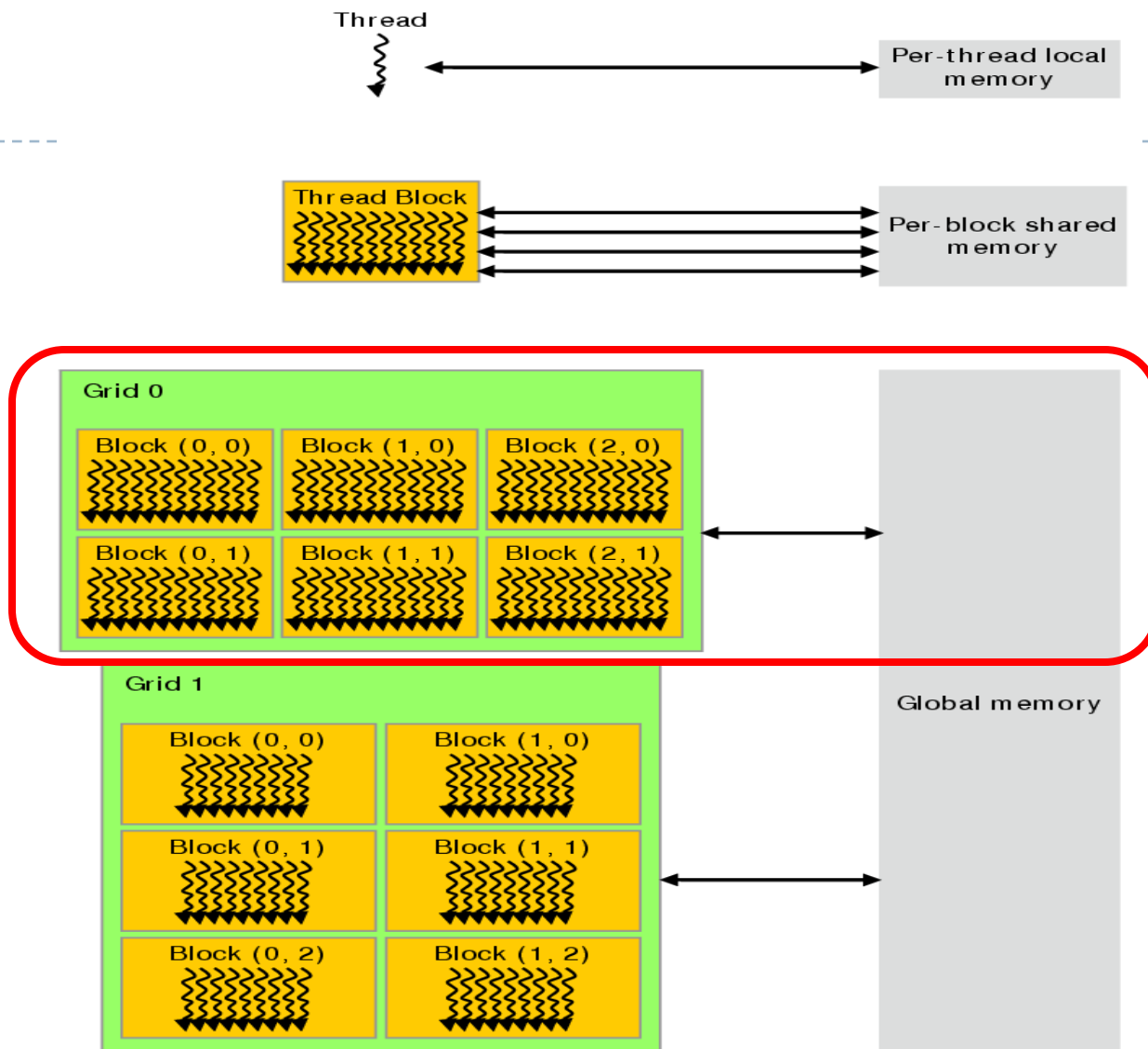


Figure 2-2. Memory Hierarchy

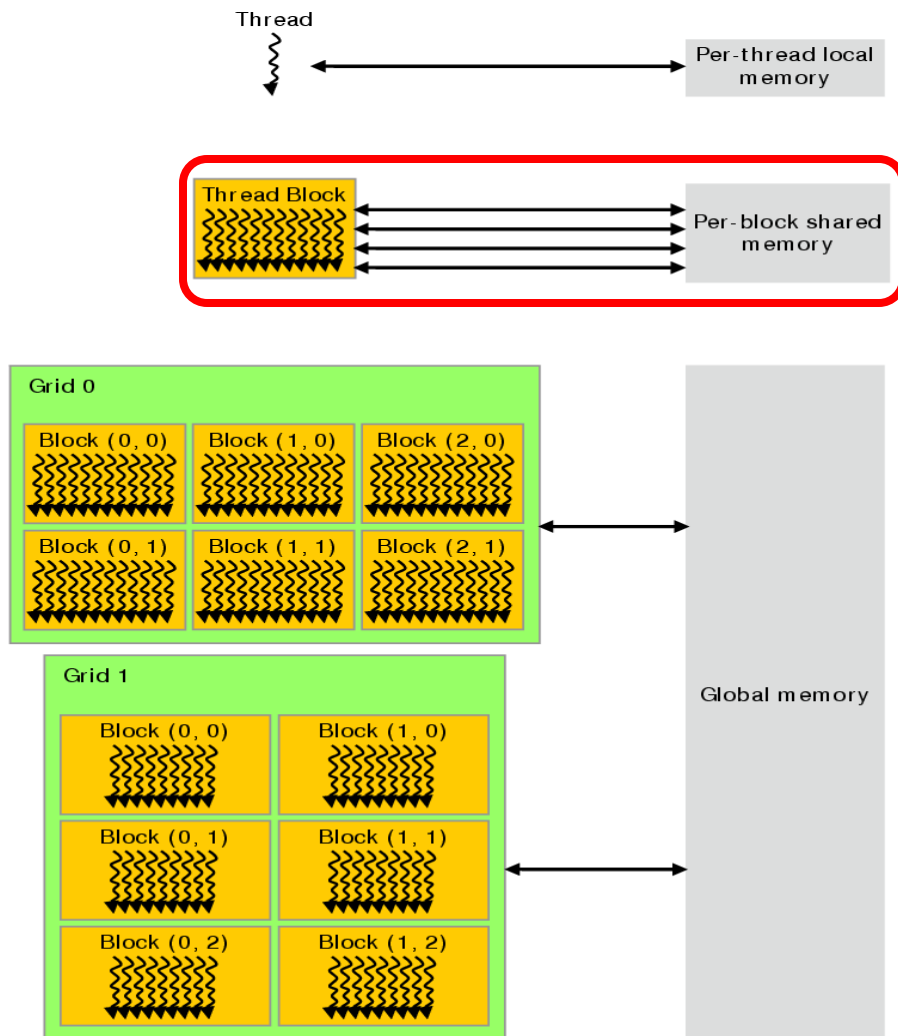
Outline (cont.)

- ▶ Assign thread blocks again ...



*Partitioning of the two subsequences ... more details later!
Thread block synchronization still needed between thread block 1 and 2
and between thread block 3 and 4!*





Now two of the subsequences can be assigned thread blocks so they can work alone.

No thread block synchronization needed. Run entirely on the GPU.

Figure 2-2. Memory Hierarchy

Outline (cont.)

- ▶ Go ahead and sort the small subsequences using one thread block on each subsequence. The right subsequence still needs two thread blocks.



Sorting ... more details later!

1 2 3 4 5 7 8 9 10 11 12 14 16 17 18 20

To Conclude:

Divide the Algorithm into two phases

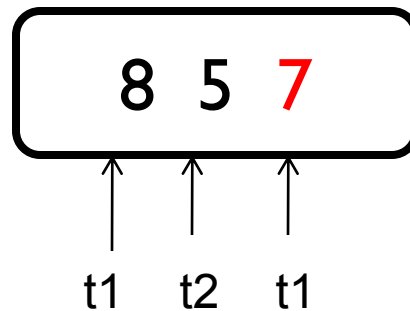
- ▶ **Phase 1:** Several thread blocks are working on the same (sub)sequence. Synchronization of the thread blocks needed.
- ▶ **Phase 2:** When there are enough subsequences available the thread blocks can work alone.
- ▶ As we shall see, these two phases are quite similar.

GPU-Quicksort Phase 2

- ▶ Assume now that each thread block has a complete subsequence to work with.
- ▶ If this subsequence is smaller than some threshold then we can use a different, faster algorithm (paper implementation used Bitonic Sort).
- ▶ Otherwise ...

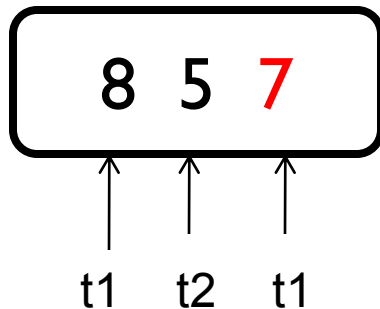
GPU-Quicksort Phase 2 (cont.)

- ▶ Assume: **One** subsequence, **One** thread block, **Two** threads in the thread block (t1,t2)



GPU-Quicksort Phase 2 (cont.)

- ▶ Go through the array in two passes.
- ▶ **Pass I:** Each thread counts how many elements it sees that are larger and smaller than the pivot element.



t1

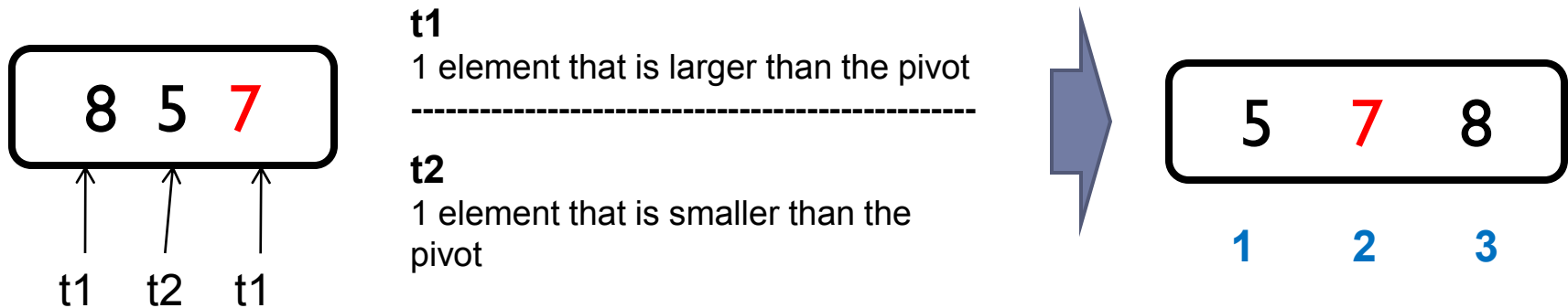
1 element that is larger than the pivot

t2

1 element that is smaller than the pivot

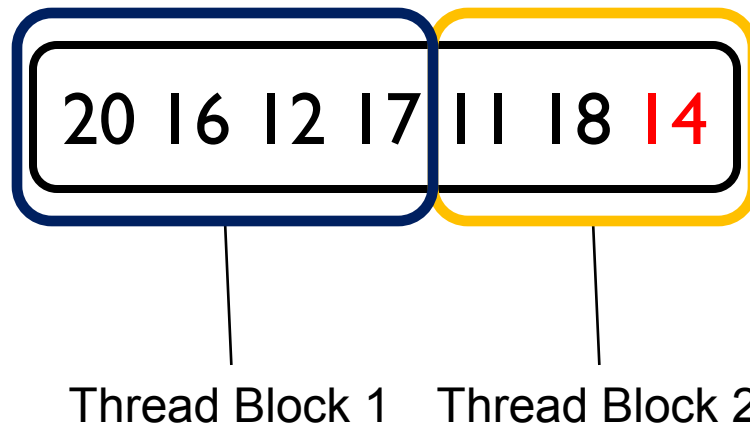
GPU-Quicksort Phase 2 (cont.)

- ▶ **Pass 2:** Use these cumulative sums to write to main memory.
 - Since there is no thread with a lower ID than thread 1 that wants to write to the **right** of the pivot, thread 1 can write to position $3 - 0 = 3$.
 - There is one thread with a lower ID than thread 2 but this thread does not want to write to the **left** of the pivot so thread 2 can write its element to position $0 + 1 = 1$.
 - One thread has to write the pivot element to memory as well!



GPU-Quicksort Phase 1

- ▶ Two thread blocks working together on the same sequence.



GPU-Quicksort Phase 1 (cont.)

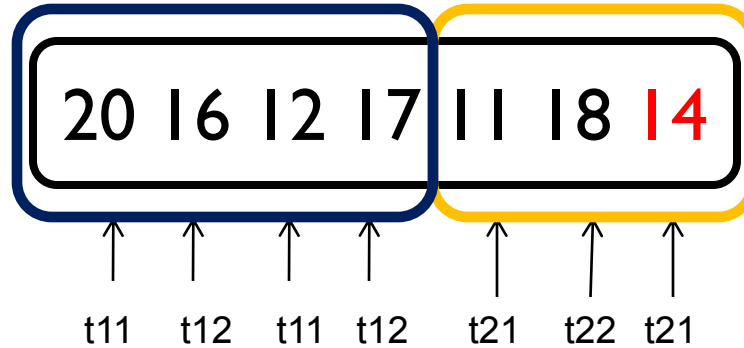
▶ Like in phase 2 **but ...**

□ In a given thread block, if a thread wants to write say to the left of the pivot it needs to know:

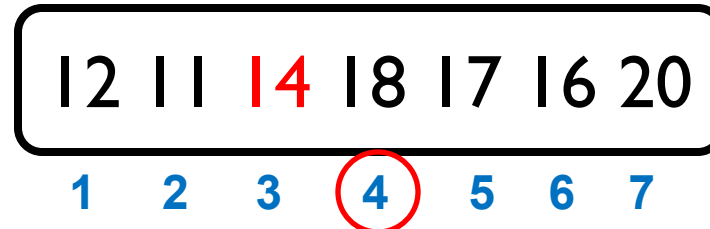
a.) How many threads (and how many elements) with lower thread ID than me are going to write to the left of the pivot in **my thread block?**

b.) How many elements **from other thread blocks (with a lower thread block ID than me)** will be written to the left of the pivot?

GPU-Quicksort Phase 1 (cont.)



*Thread t22 wants to write element 18 to the **right** of the pivot. There are three elements that thread block 1 wants to write to the **right** but zero threads in thread block 2 that wants to write to the **right**. So thread t22 writes to position $7 - 3 = 4$. And so on ...*

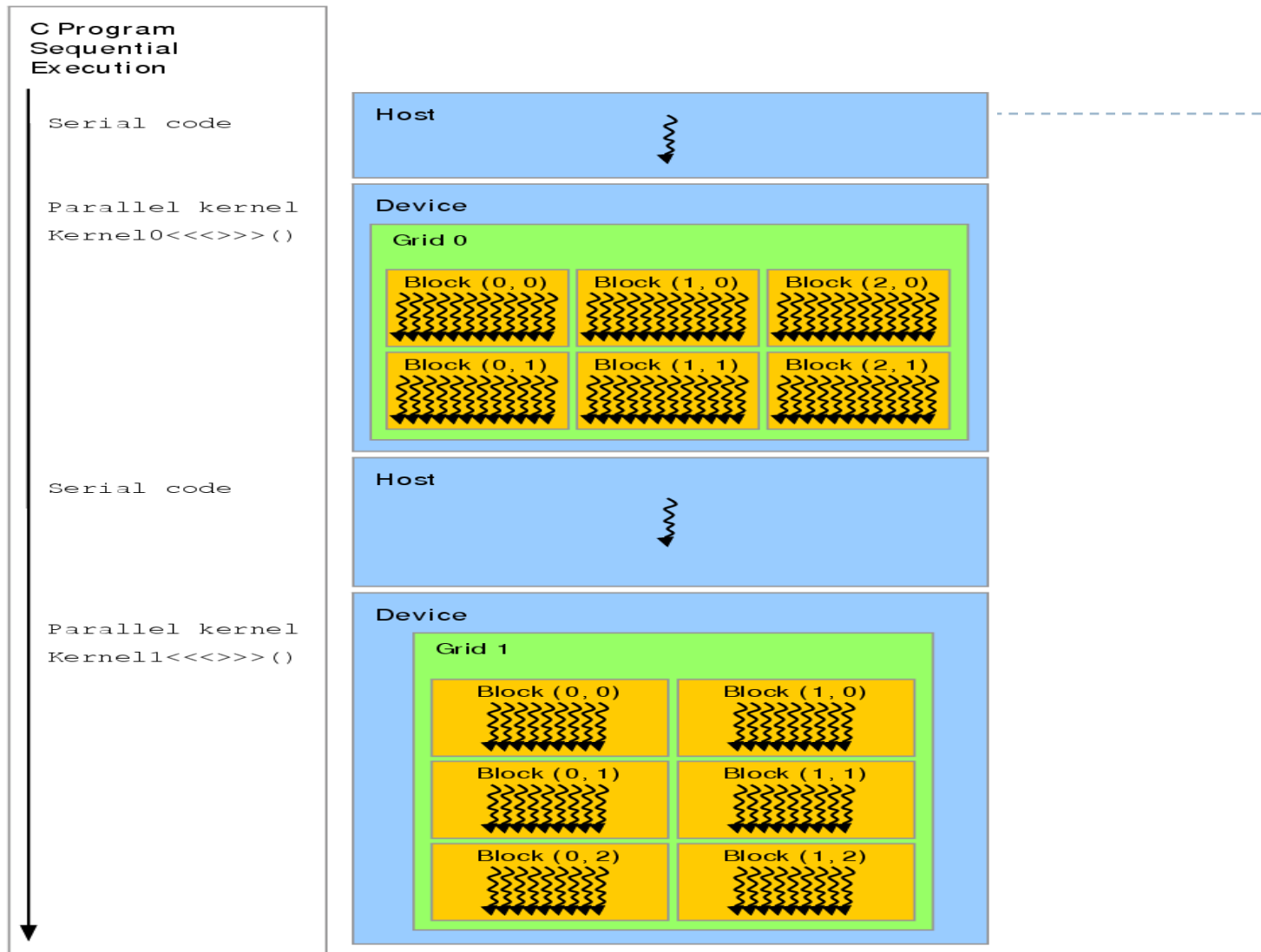


Summary

- ▶ **Two phases:**
 - ▶ In the first phase we need thread block synchronization.
 - ▶ In the second phase we have enough subsequences so that each thread block can work “on its own”.
- ▶ We calculate cumulative sums in both phases so that we know how to place the data in global memory.
- ▶ We don't do the sorting in-place. Instead we use an auxiliary array. So we have two arrays that we swap data between.

Summary (cont.)

- ▶ Data is read in chunks of T words, where T is the number of threads in each thread block (coalescing of reads).
- ▶ For calculating the cumulative sum we can use atomic primitive functions such as Fetch-And-Add (not available on all GPUs).
- ▶ Thread block barrier functions needed in the first phase.



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

Measurements

Measurements

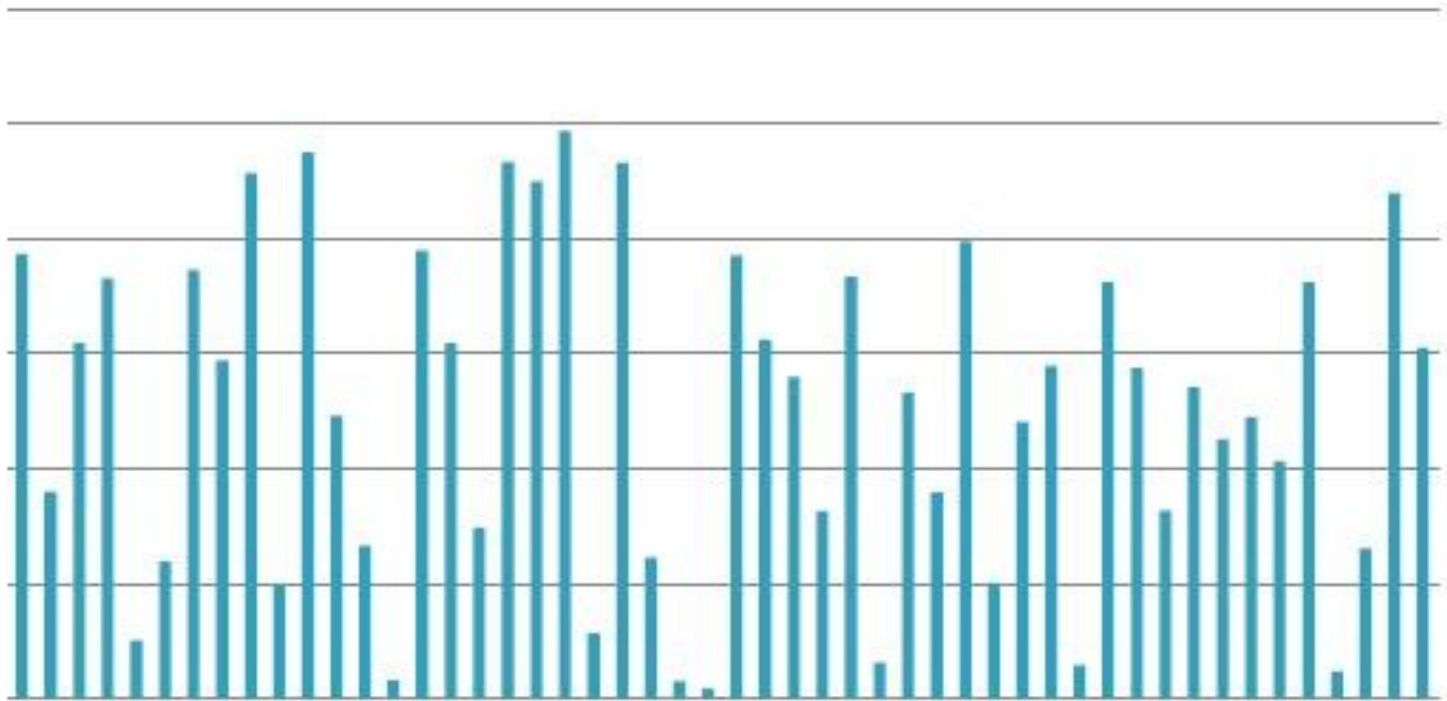
- ▶ GPU-Quicksort Cederman and Tsigas, ESA08
- ▶ GPUSort Govindaraju et. al., SIGMOD05
- ▶ Radix-Merge Harris et. al., GPU Gems 3 '07
- ▶ Global radix Sengupta et. al., GH07
- ▶ Hybrid Sintorn and Assarsson, GPGPU07

- ▶ Introsort David Musser, Software: Practice and Experience

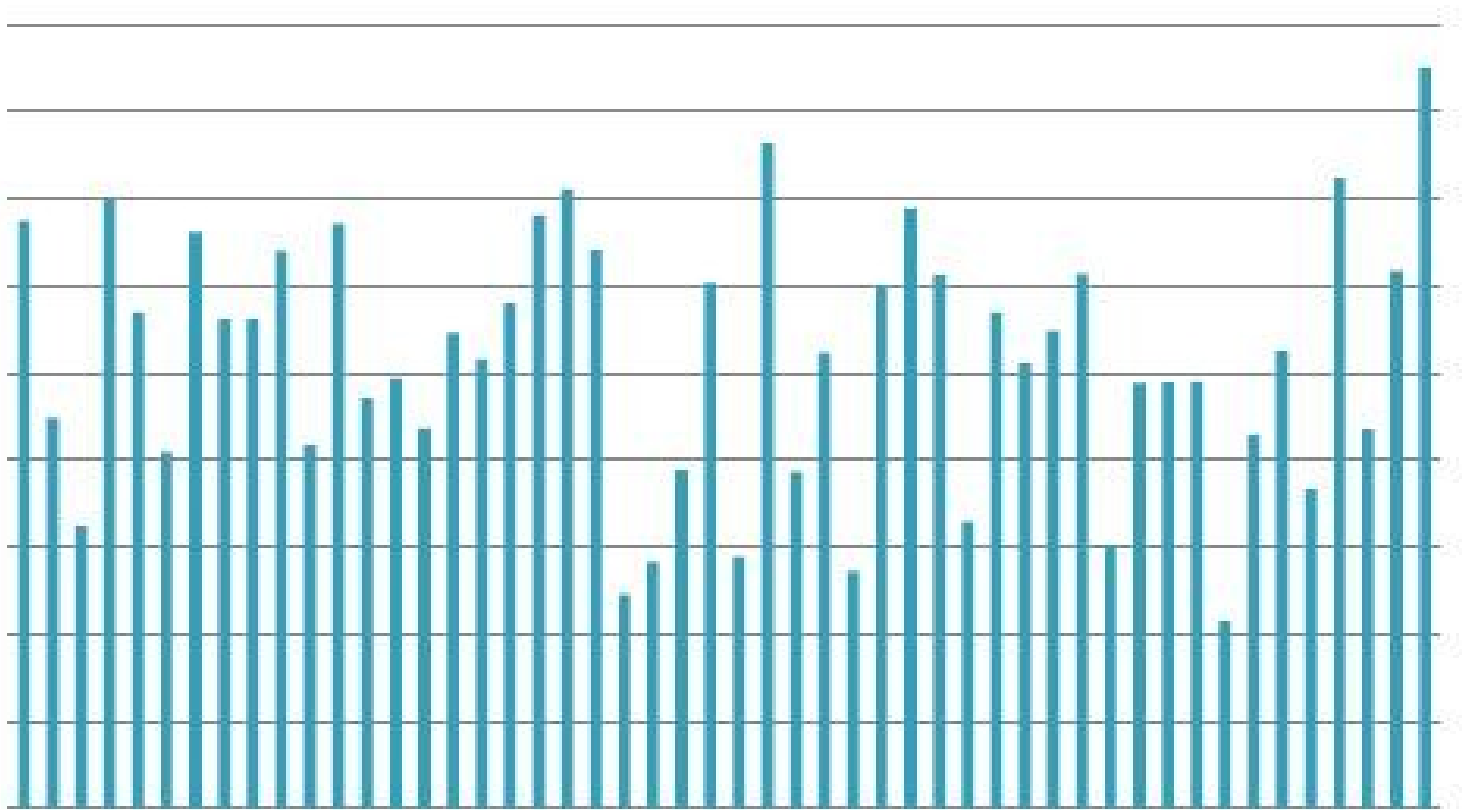
Distributions

- ▶ Uniform
- ▶ Gaussian
- ▶ Bucket
- ▶ Sorted
- ▶ Zero
- ▶ Stanford Models

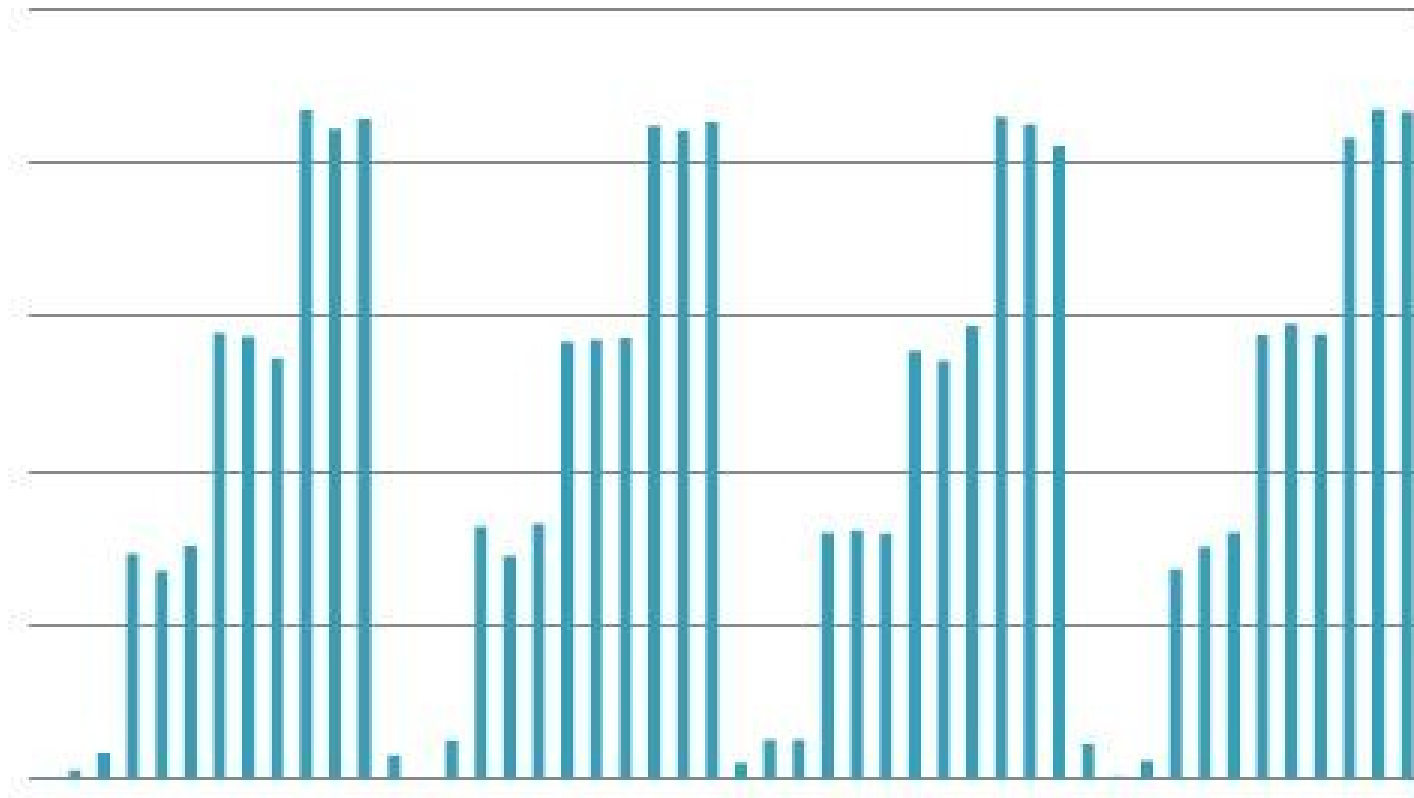
► Uniform



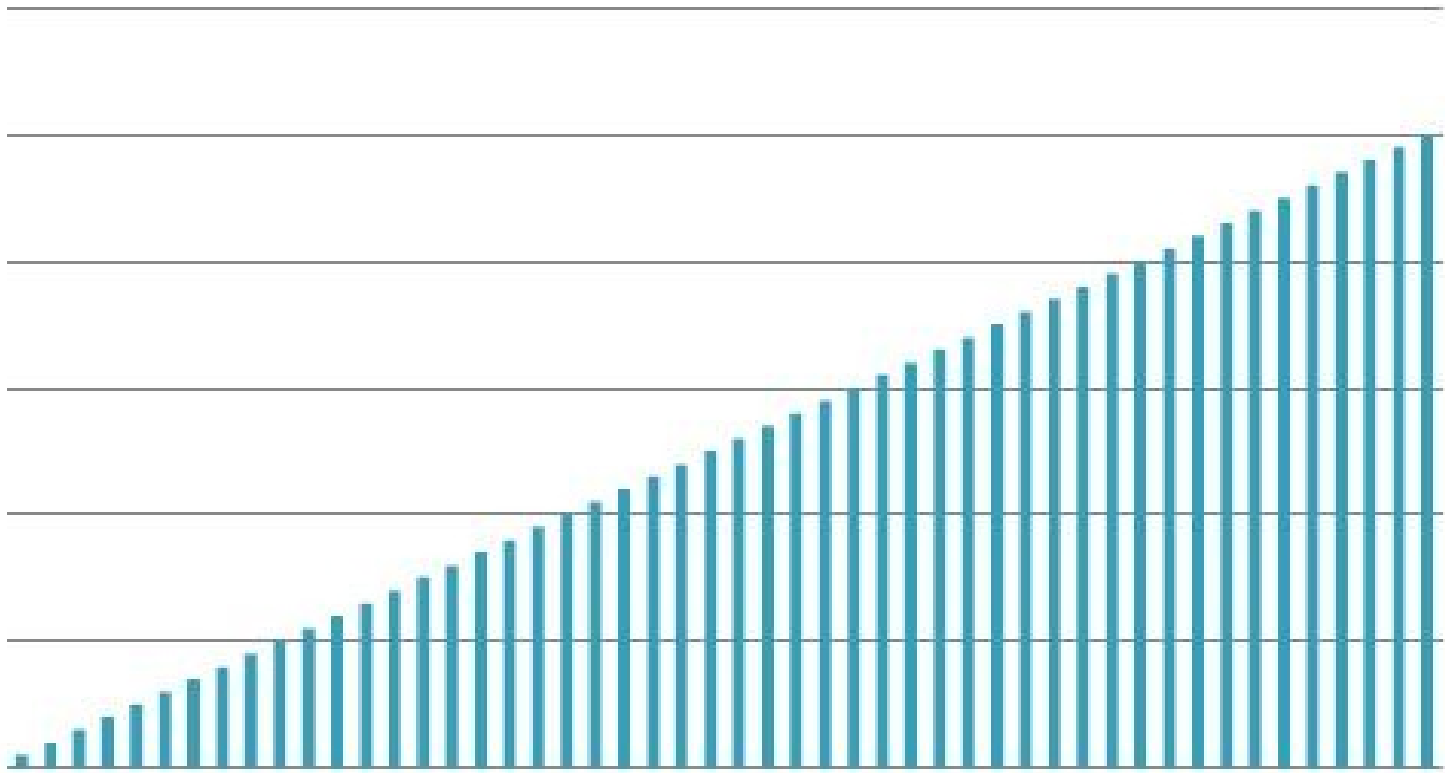
► Gaussian



► **Bucket**



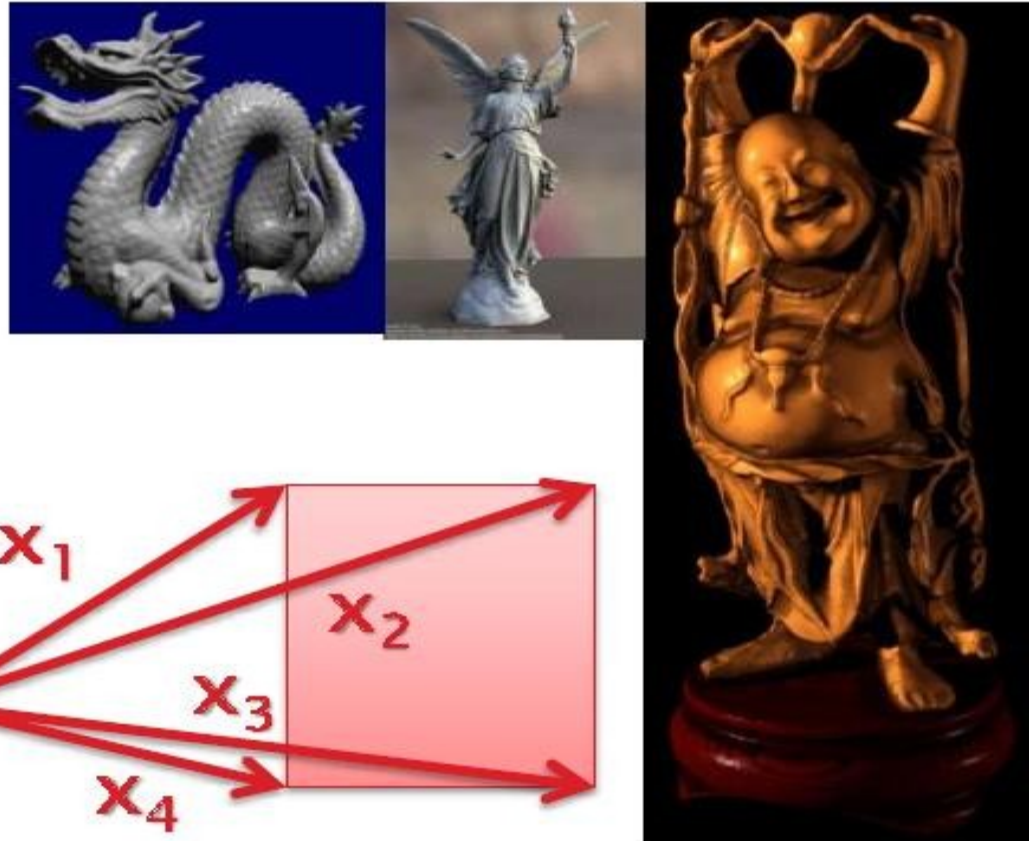
▶ Sorted



► Zero



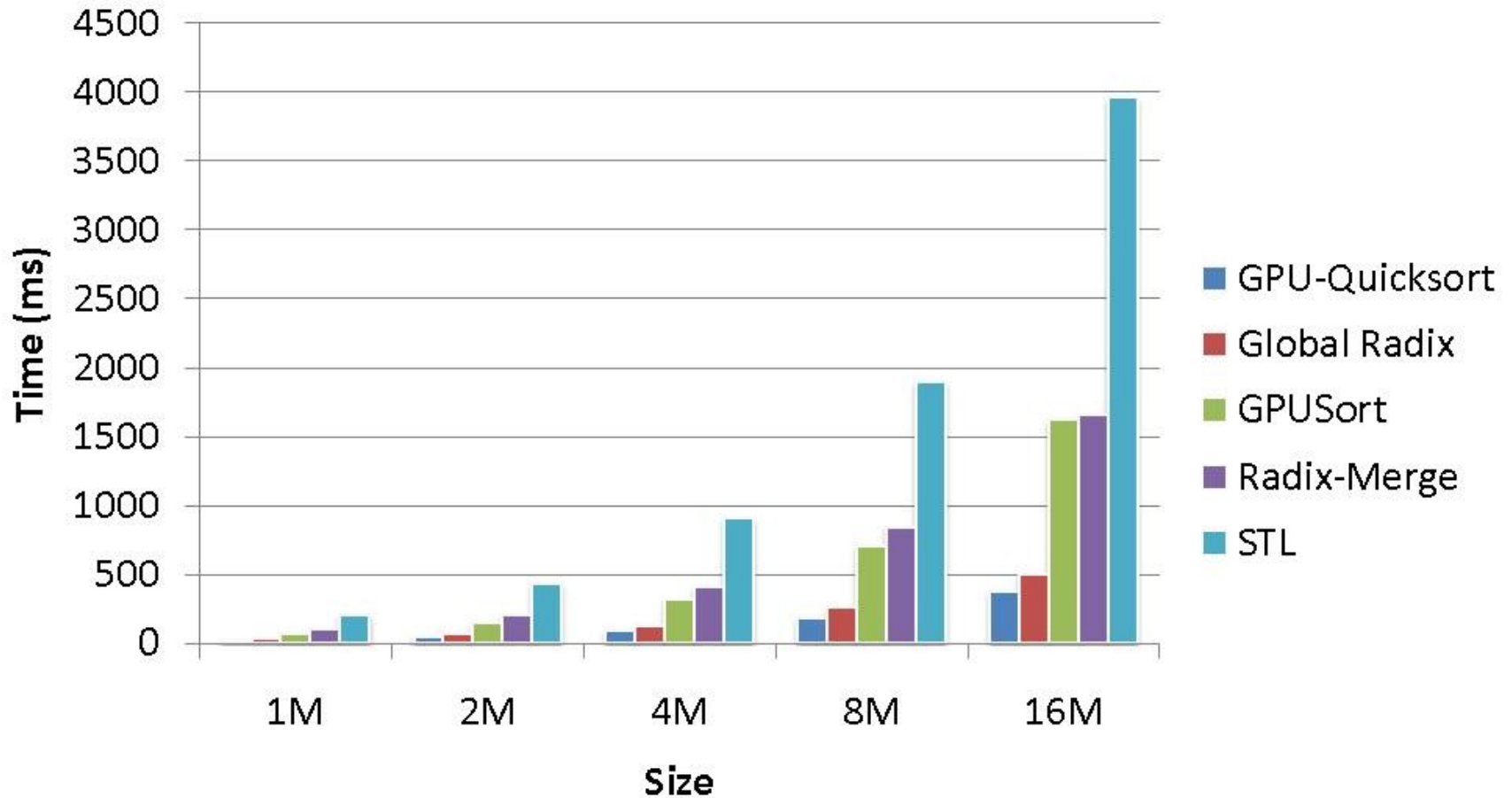
► **Stanford Models**



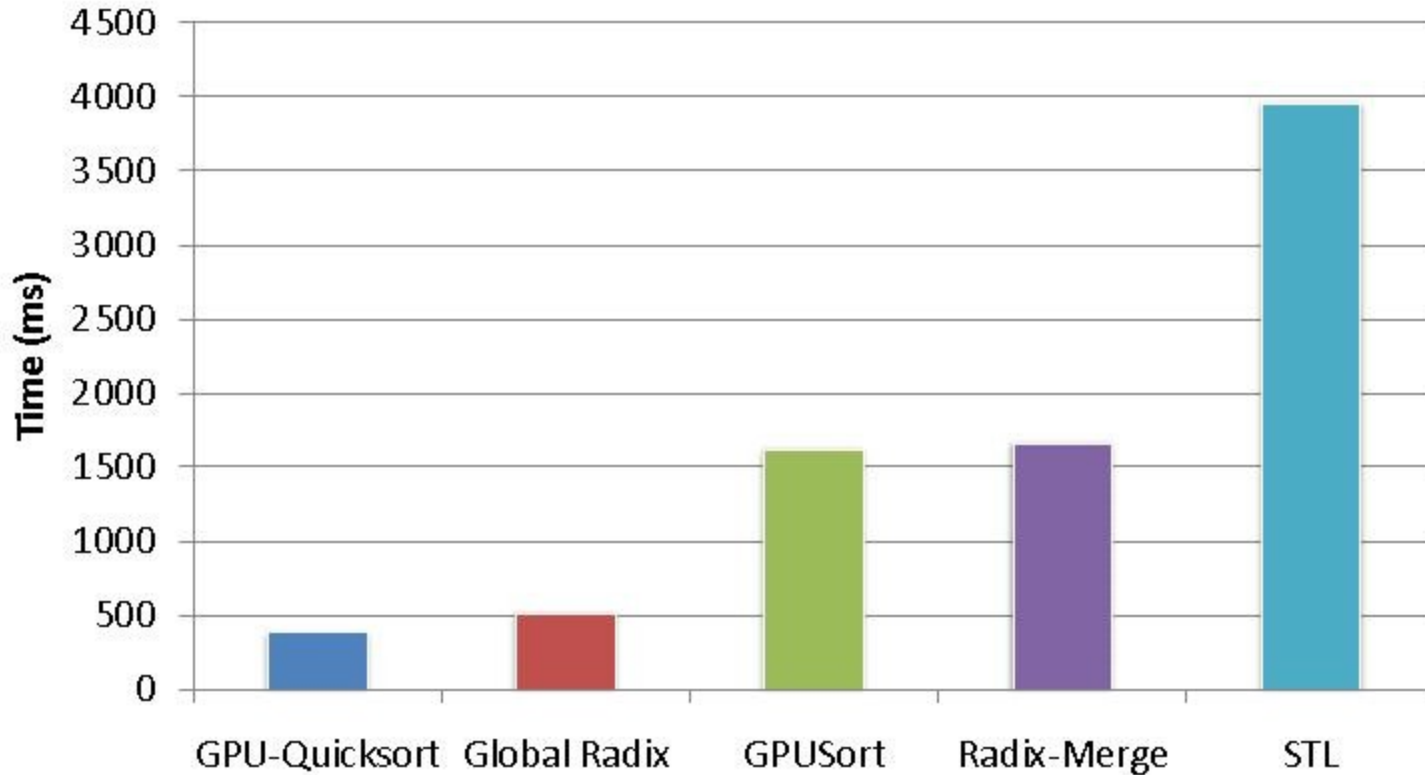
Hardware

- ▶ **8800GTX**
 - ▶ 16 multiprocessors
 - ▶ 128 stream processor cores
 - ▶ 768MB memory account
 - ▶ 86.4 GB/s bandwidth
- ▶ **8600GTS**
 - ▶ 4 multiprocessors
 - ▶ 32 stream processor cores
 - ▶ 256MB memory account
 - ▶ 32 GB/s bandwidth
- ▶ **CPU-Reference**
 - ▶ AMD Dual-Core Opteron 265 / 1.8 GHz

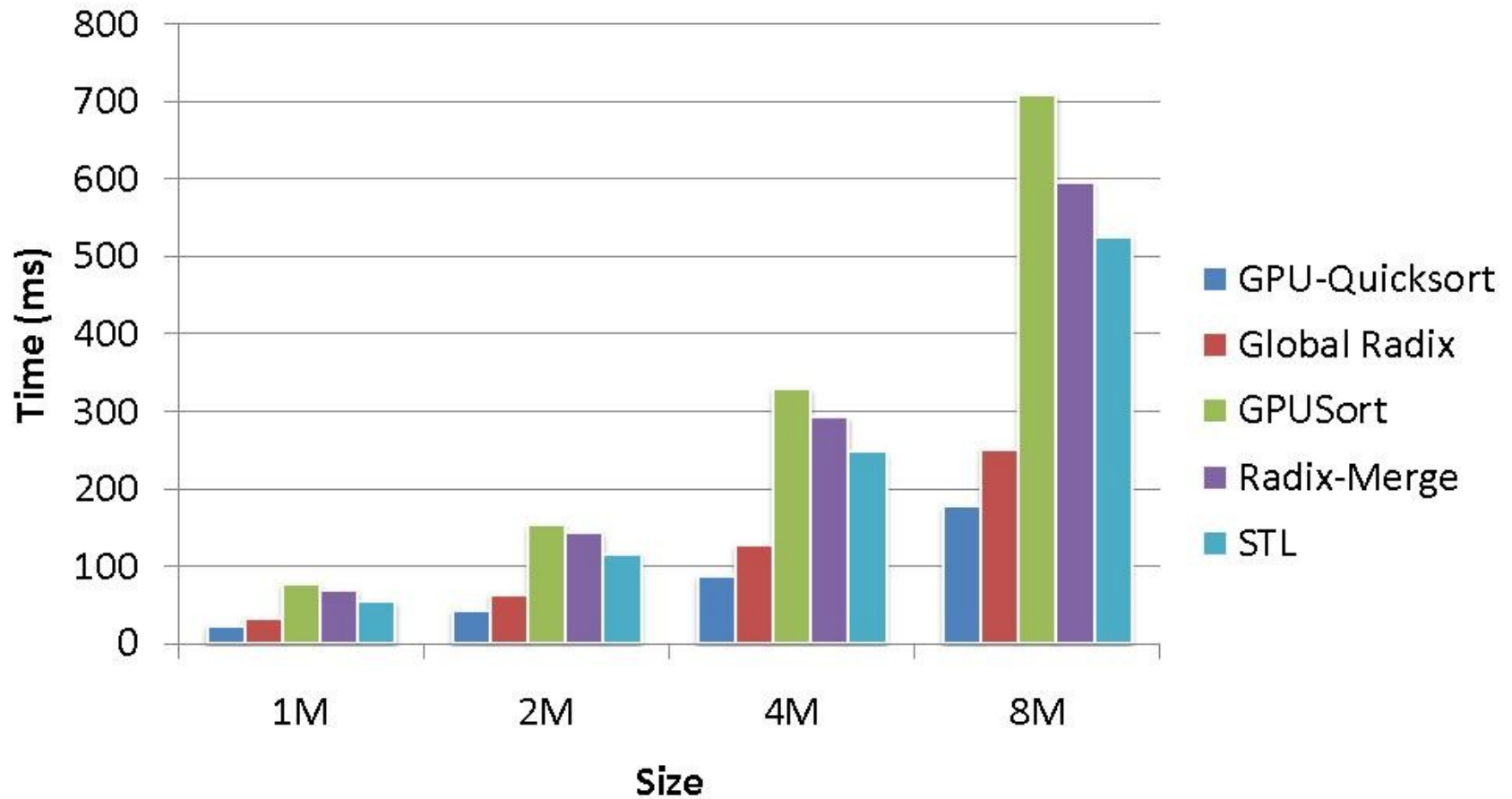
8800GTX – Uniform Distribution



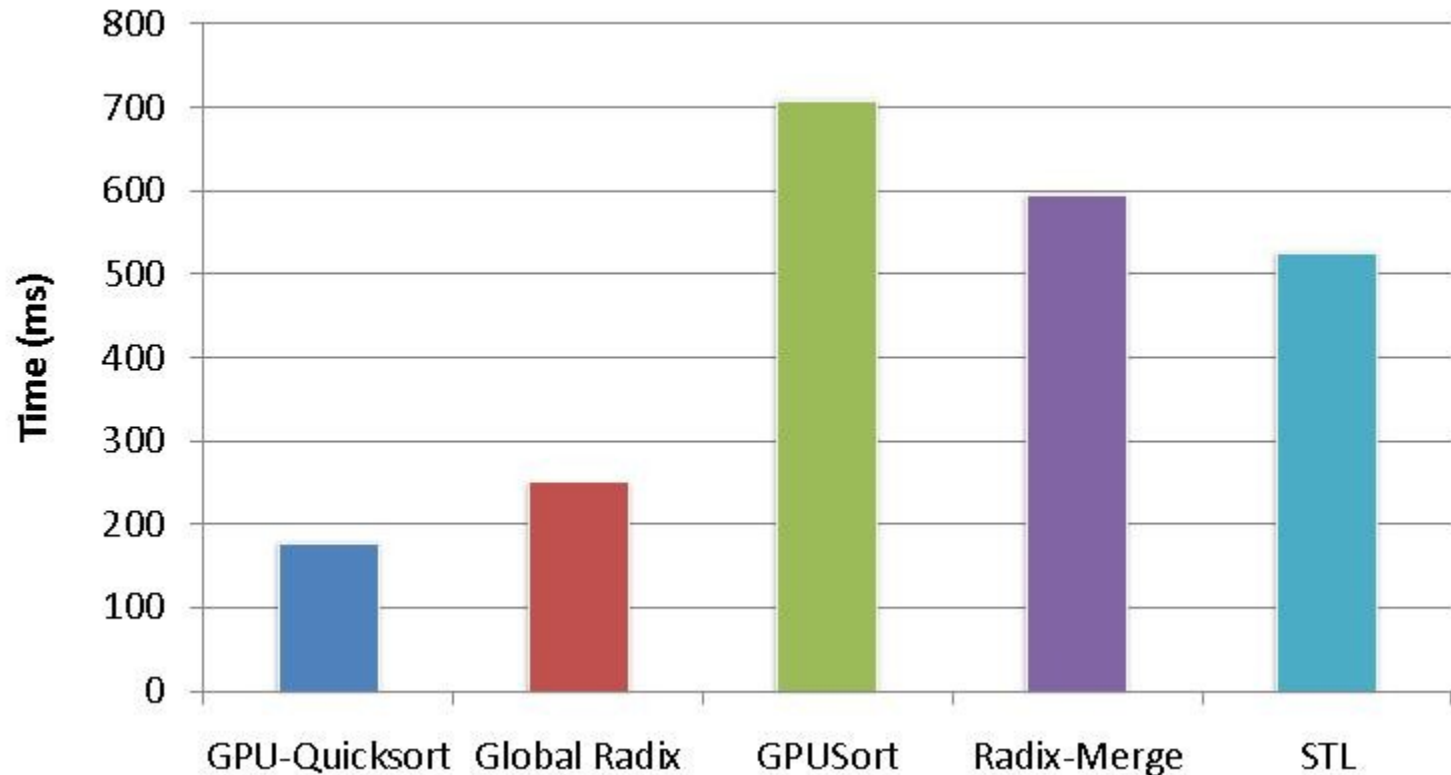
8800GTX – Uniform Distribution 16MB



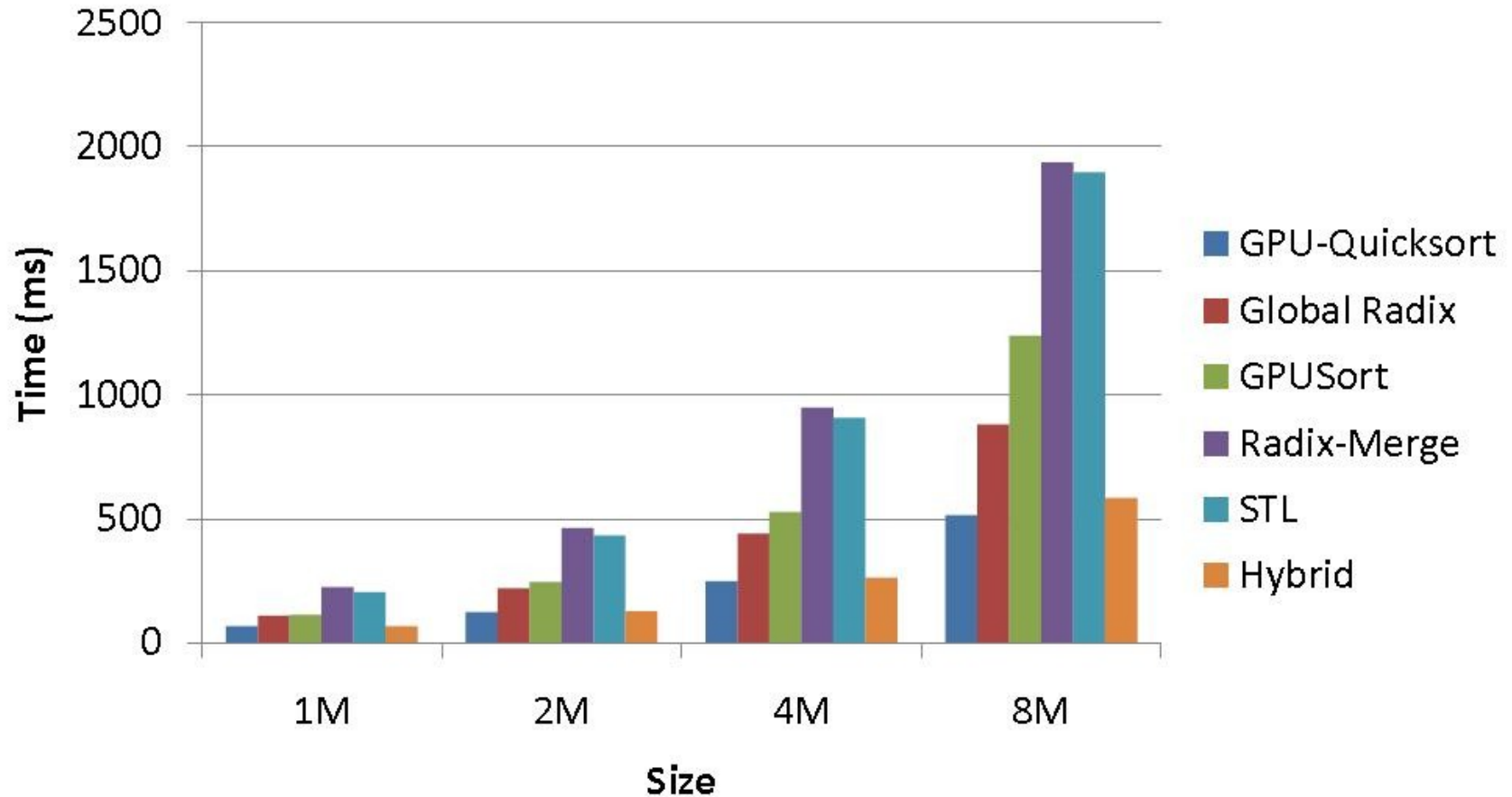
8800GTX – Sorted Distribution



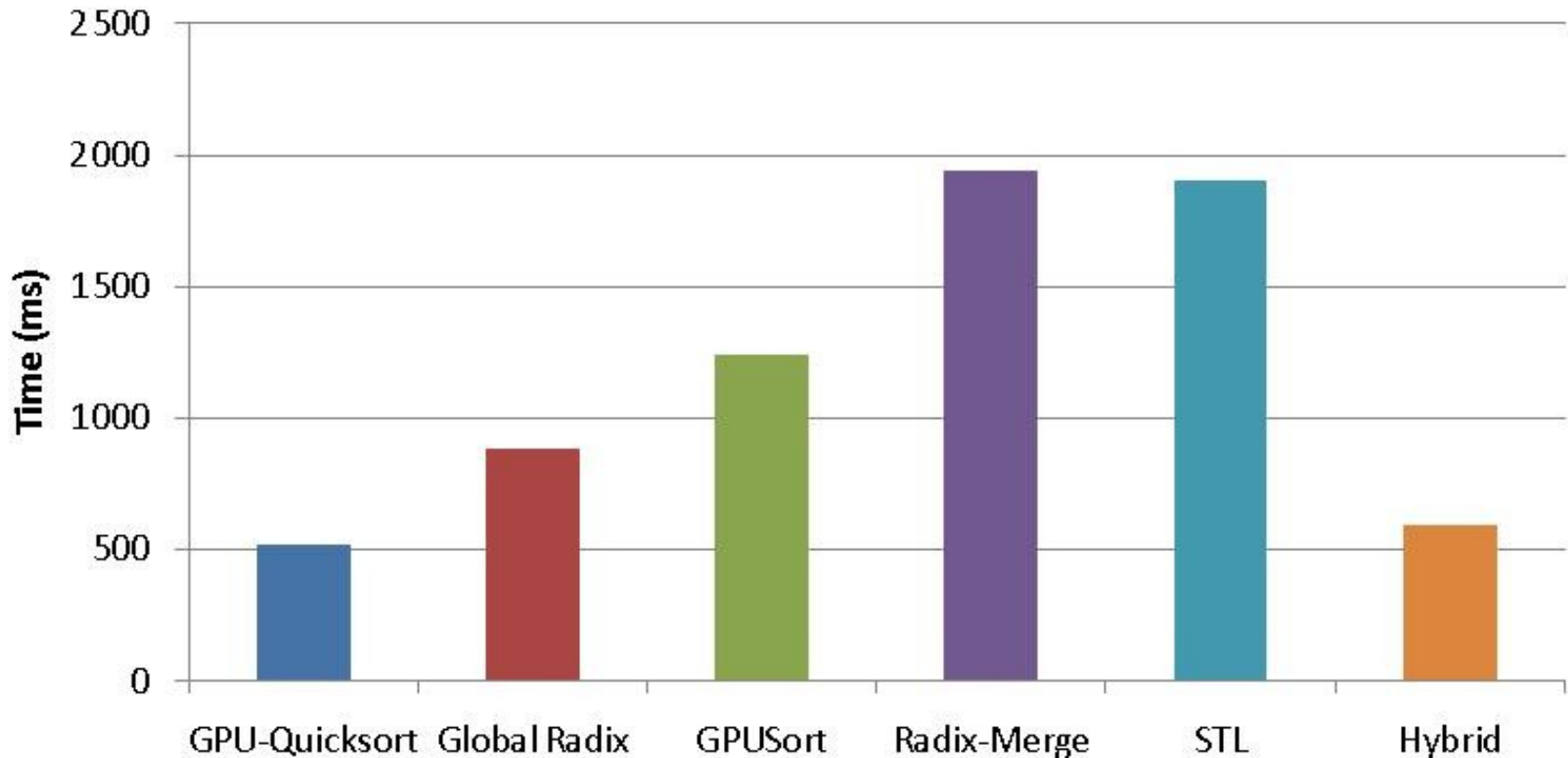
8800GTX – Sorted Distribution 8M



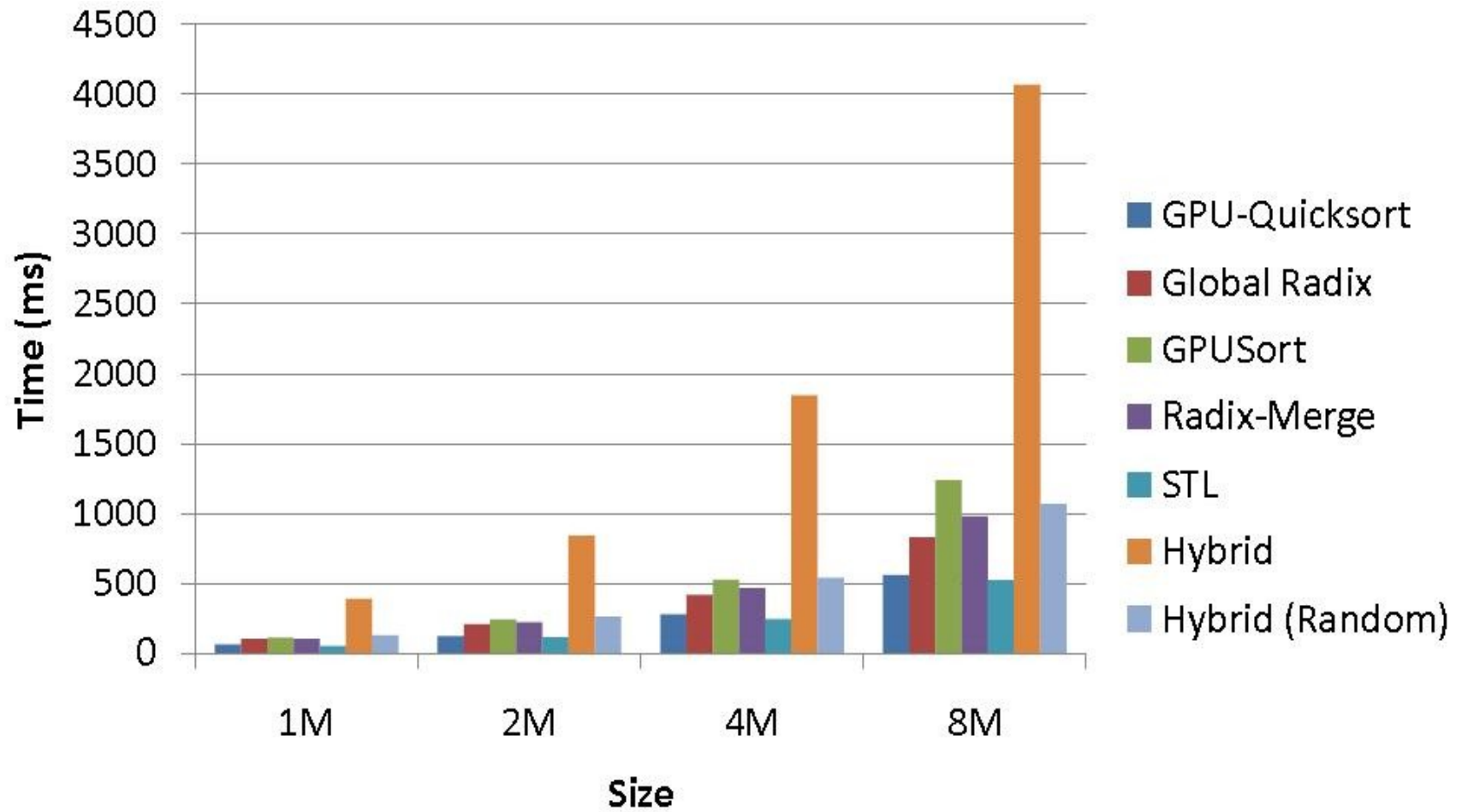
8600GTS – Uniform Distribution



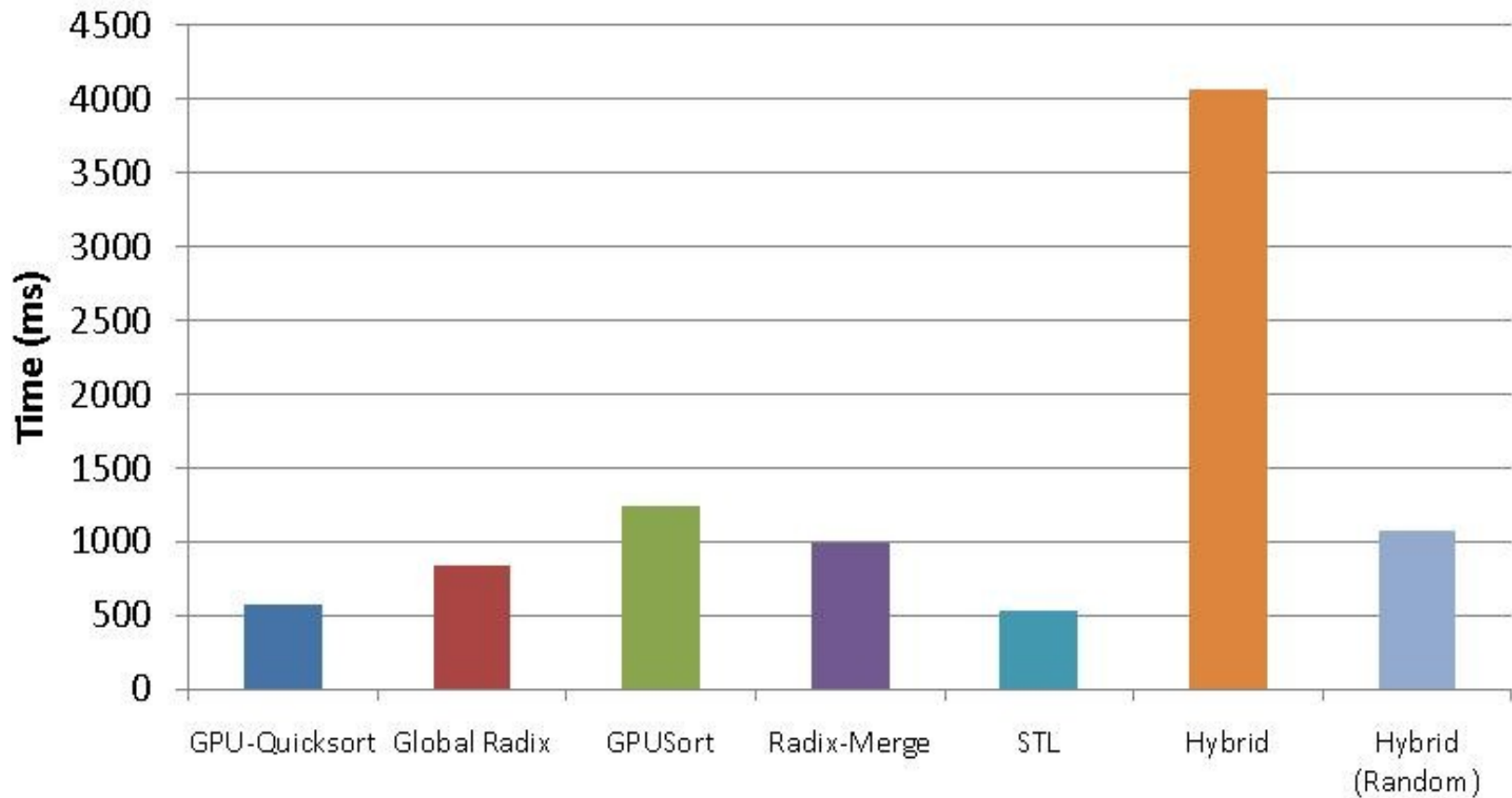
8600GTS – Uniform Distribution 8M



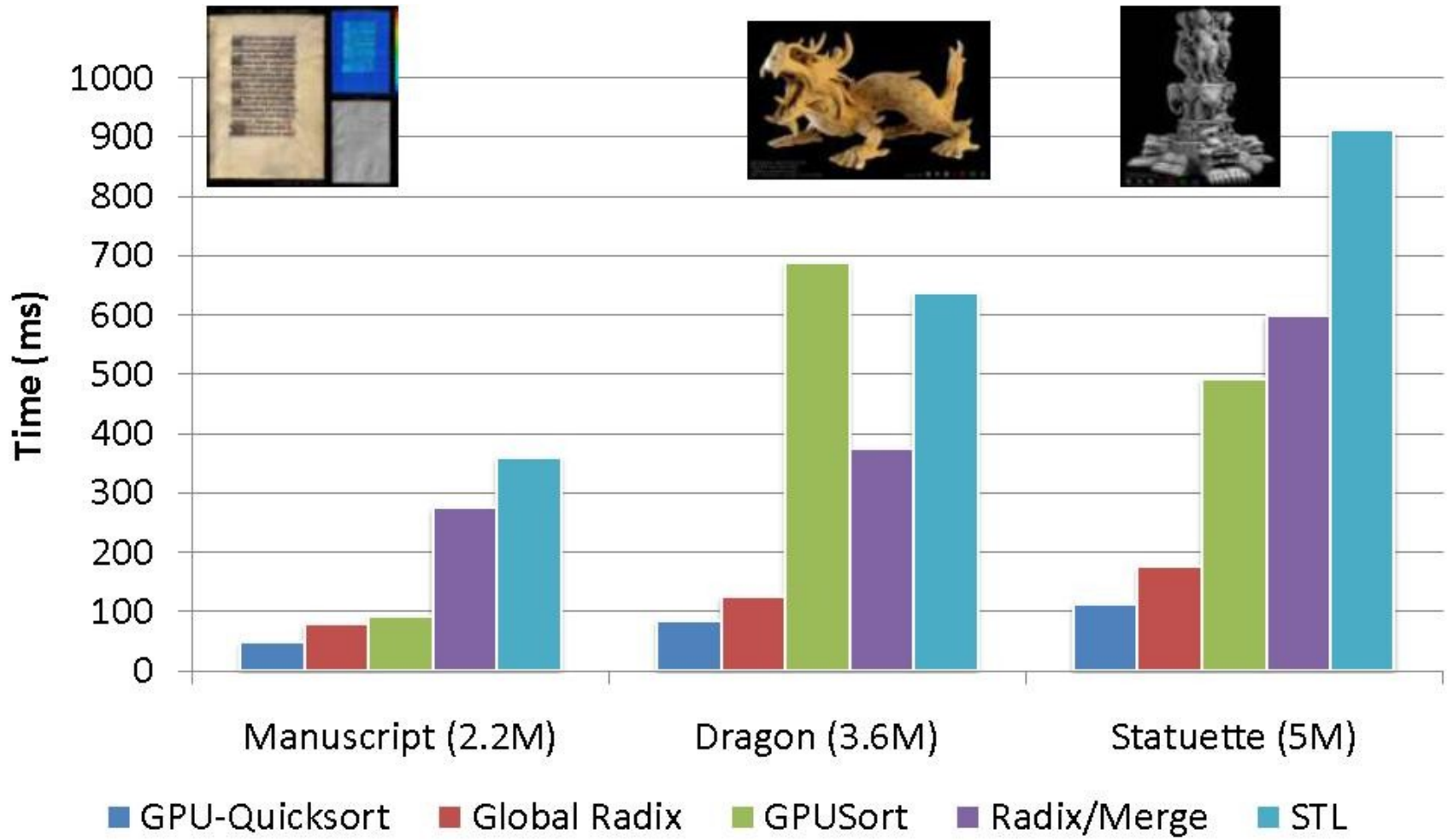
8600GTS – Sorted Distribution



8600GTS – Sorted Distribution 8M



8800GTX – Visibility Ordering



Conclusions

Conclusions

- ▶ Minimal, manual cache
 - ▶ Used only for prefix sum and bitonic sort
- ▶ 32-word SIMD instruction
 - Main part executes same instructions
- ▶ Coalesced memory access
 - All reads coalesced
- ▶ Block synchronization only required in the first phase
- ▶ Expensive synchronization primitives

Conclusions (cont.)

- ▶ Quicksort is a viable sorting method for graphics processors and can be implemented in a data parallel way
- ▶ It is competitive