

# Sorting and Searching

**Naga Govindaraju**  
**Microsoft Corporation**



# Topics

---

- **Sorting**
  - Sorting networks
  - External memory sorting
- **Search**
  - Searching quantiles

# Assumptions

---

- Data organized into 1D arrays
- Rendering pass == screen aligned quad
- Programmable GPU

# Sorting

---

# Sorting

---

- **Given an unordered list of elements, produce list ordered by key value**
  - Kernel: compare and swap
- **Data-parallel and data oblivious algorithms**
  - Bitonic merge sort [Batcher 68]
  - Periodic balanced sorting networks [Dowd 89]

# Bitonic Merge Sort Overview

---

- Repeatedly build bitonic lists and then sort them
  - Bitonic list is two monotonic lists concatenated together, one increasing and one decreasing.
    - List A: (3, 4, 7, 8)                      monotonically increasing
    - List B: (6, 5, 2, 1)                      monotonically decreasing
    - List AB: (3, 4, 7, 8, 6, 5, 2, 1)                      bitonic

# Bitonic Merge Sort

---

3

7

4

8

6

2

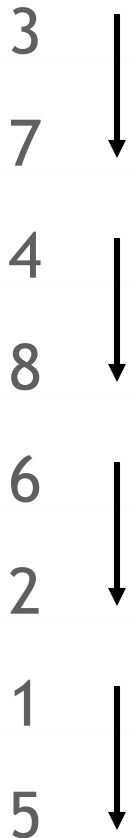
1

5

8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)

# Bitonic Merge Sort (Stage 1)

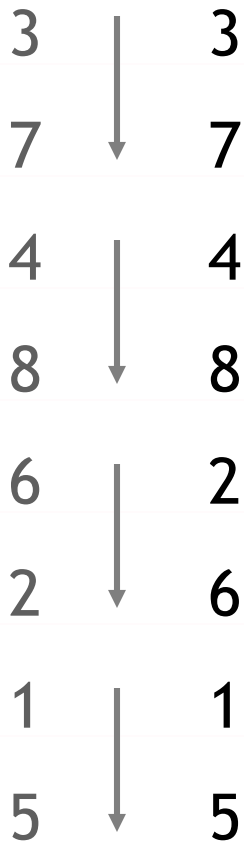
---



Each arrow defines a comparator. Minimum is stored at tail and maximum is stored at head

# Bitonic Merge Sort (End of Stage 1)

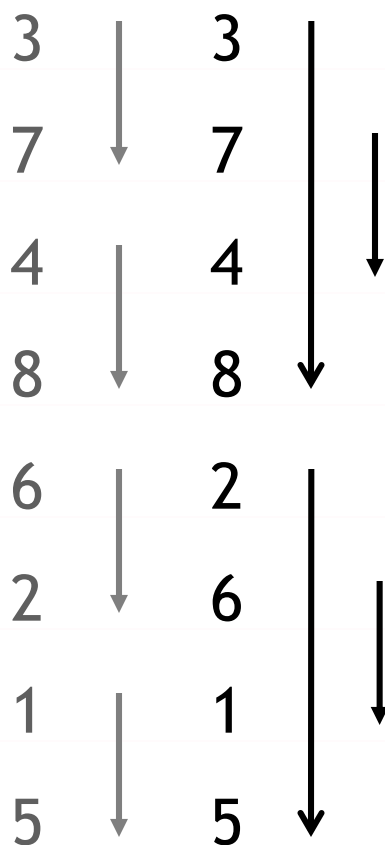
---



Stage 1 has 1 step and sorts two element lists  
4x sorted lists: (3,7) (4,8) (2,6) (1,5)

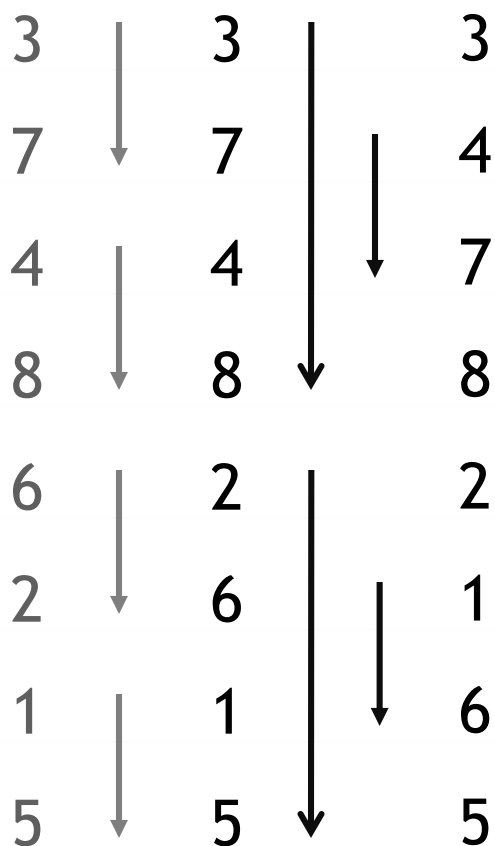
# Bitonic Merge Sort (Stage 2, Step 2)

---



Stage 2 has two steps. In this step, comparisons are performed on 4 consecutive elements

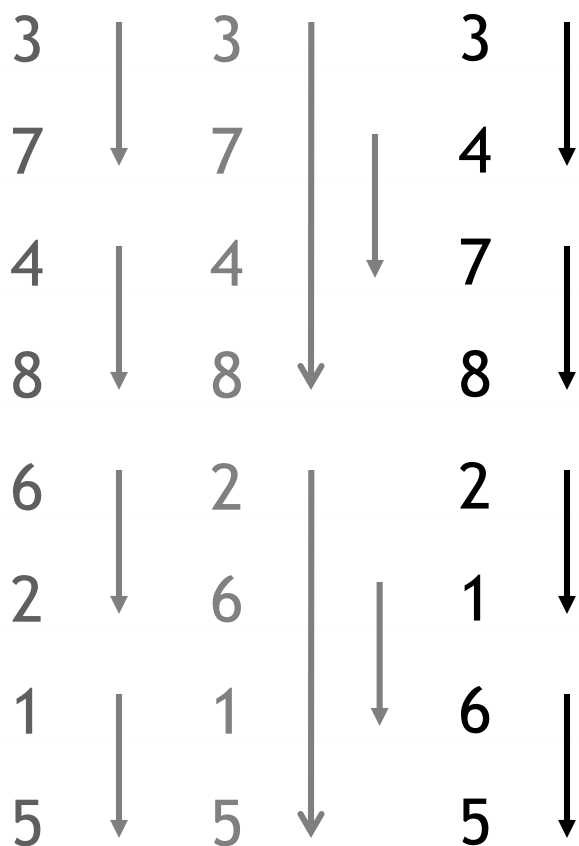
# Bitonic Merge Sort (Stage 2, Step 2)



The first step in each stage ensures that elements in first half of the list  $\leq$  elements in second half of the list. For example  $\{3,4\} \leq \{7,8\}$

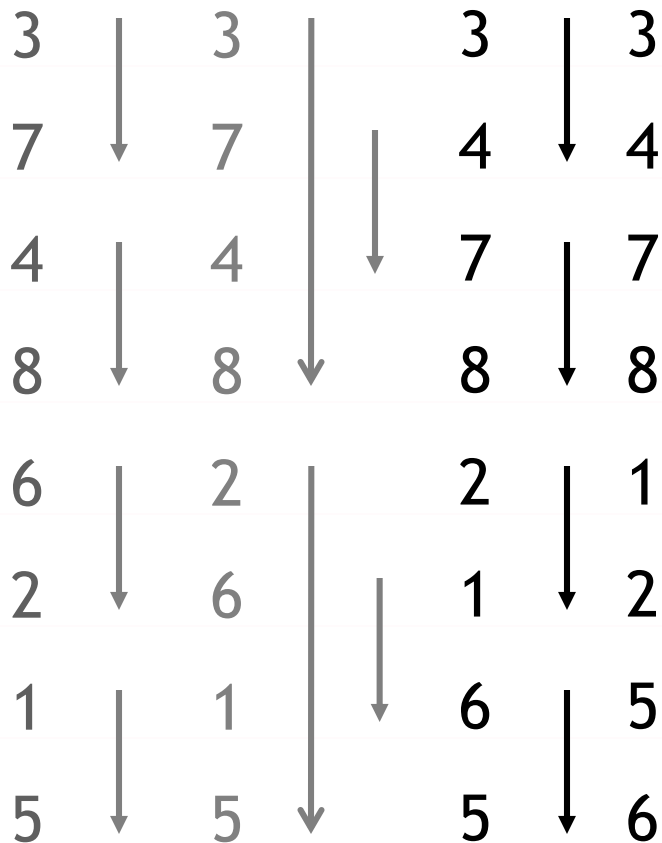
# Bitonic Merge Sort (Stage 2, Step 1)

---



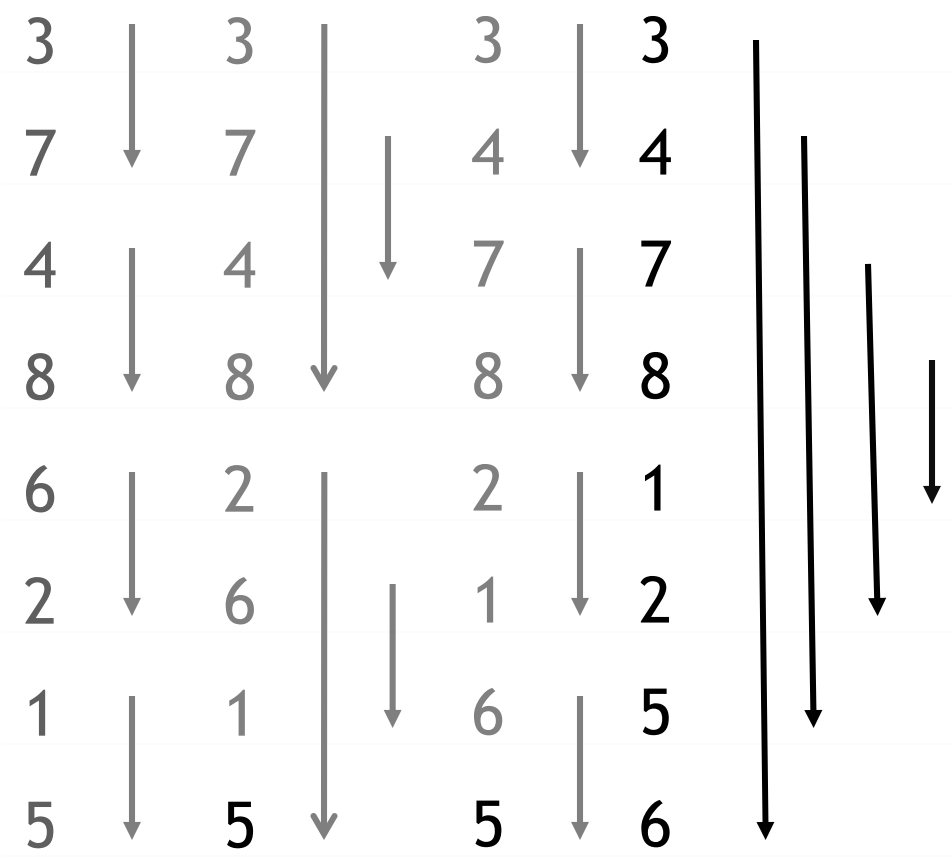
In all subsequent steps, the comparators are in the same direction - ensures good locality

# Bitonic Merge Sort (End of Stage 2)



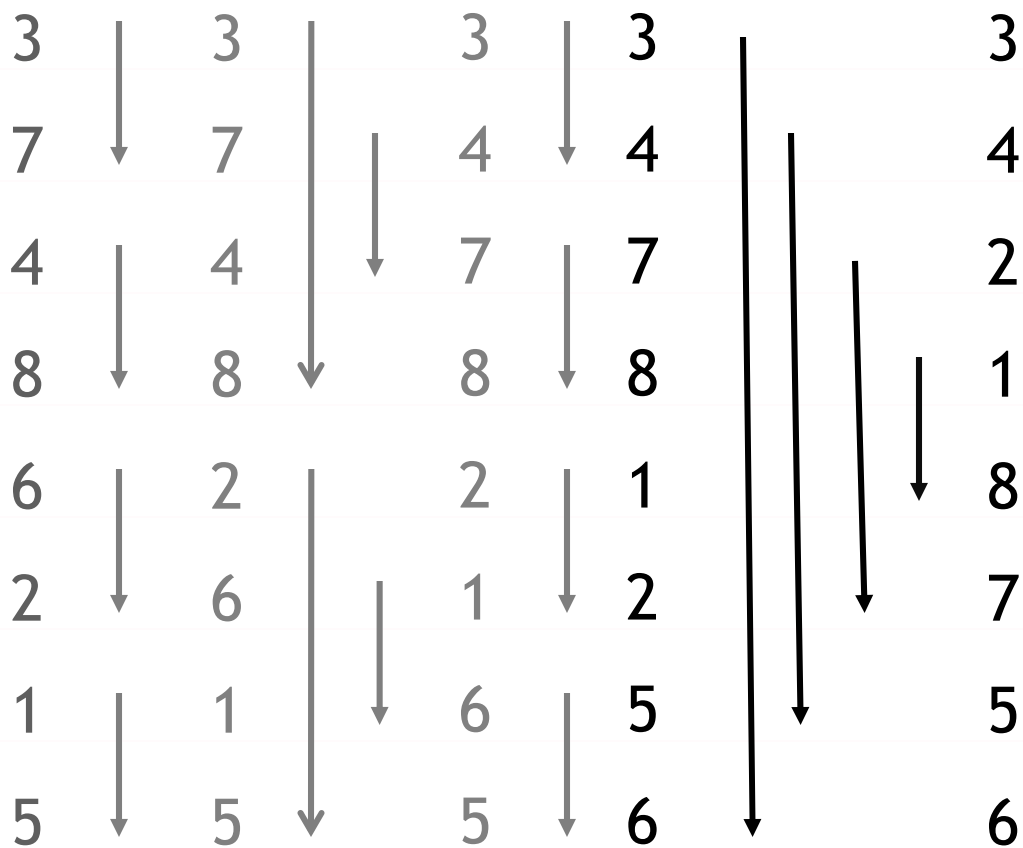
Two sorted lists of size 4 each (3,4,7,8) and (1,2,5,6)

# Bitonic Merge Sort (Stage 3, Step 3)



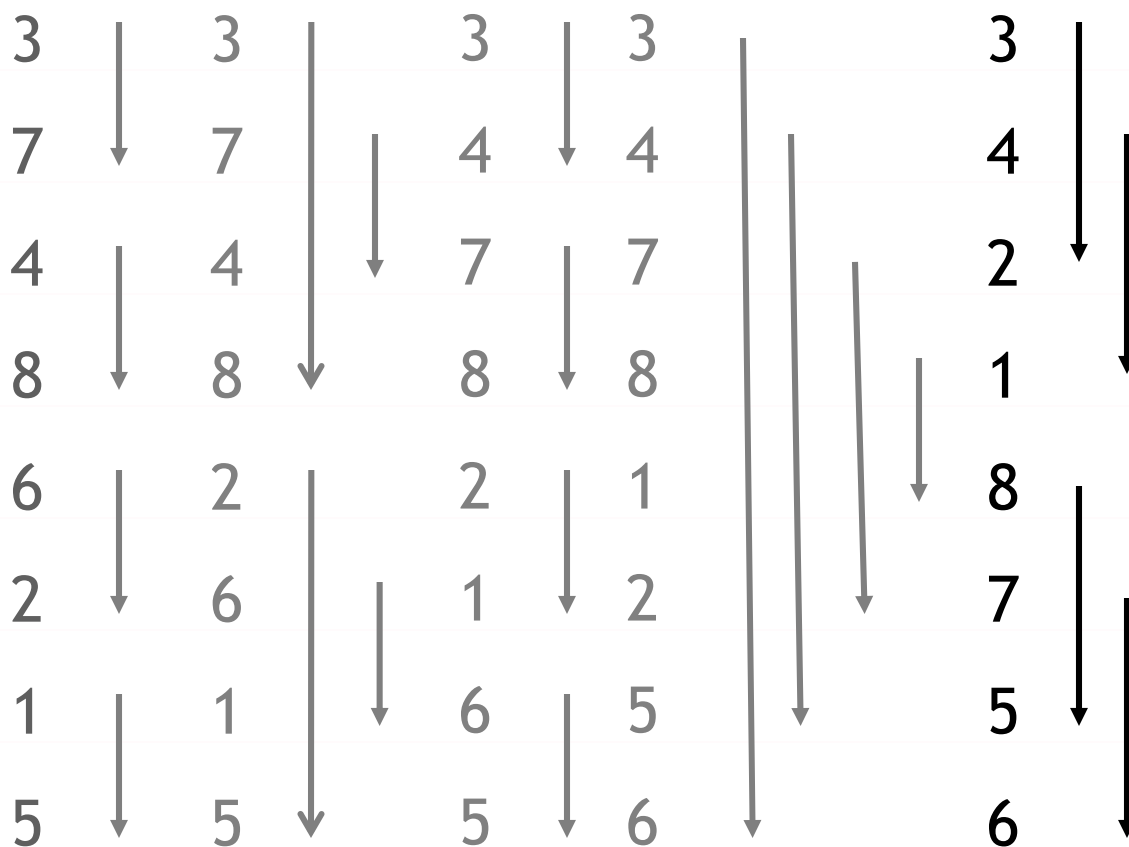
Stage 3 has three steps. Beginning step in a stage, compare the first half of the list with the reversal of next half of list - (3,4,7,8) and (6,5,2,1)

# Bitonic Merge Sort (Stage 3, Step 3)



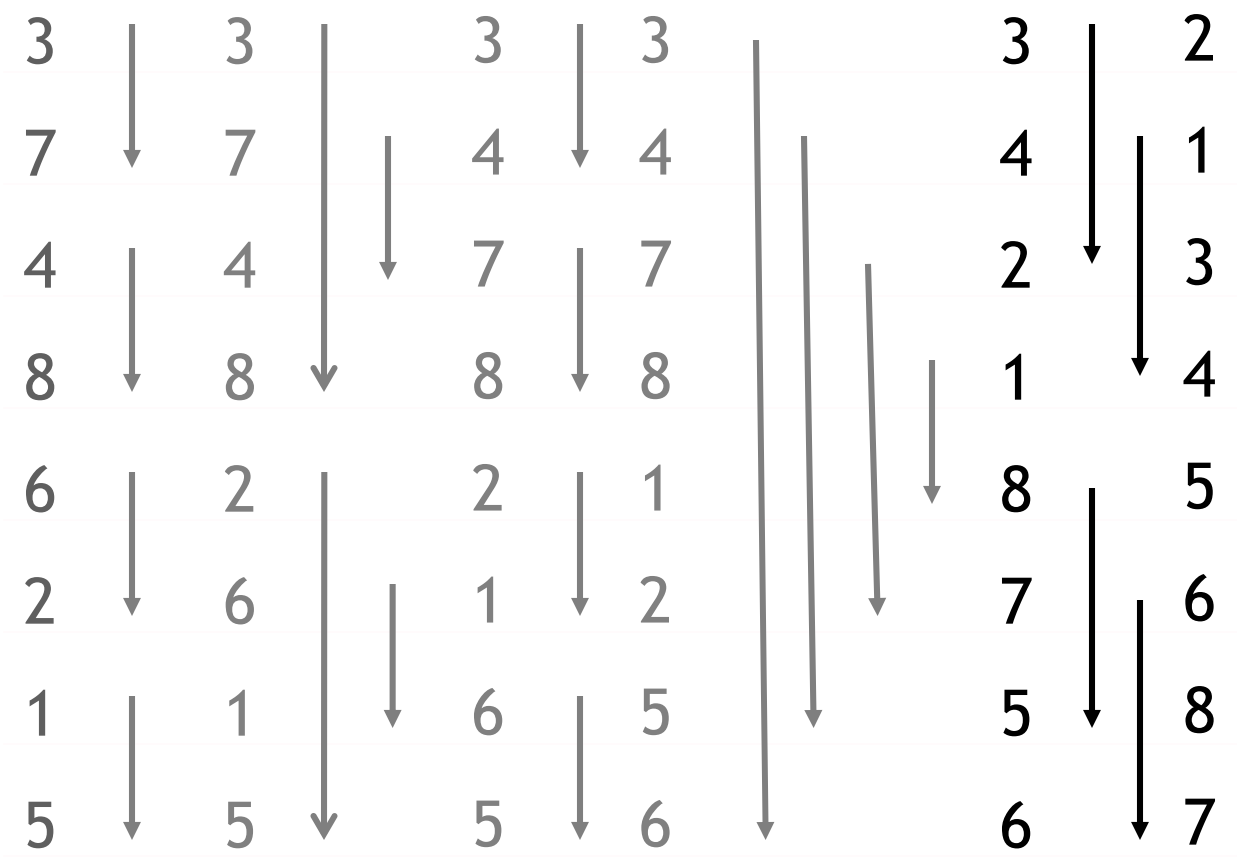
$$(3, 4, 2, 1) \leq (8, 7, 5, 6)$$

# Bitonic Merge Sort (Stage 3, Step 2)



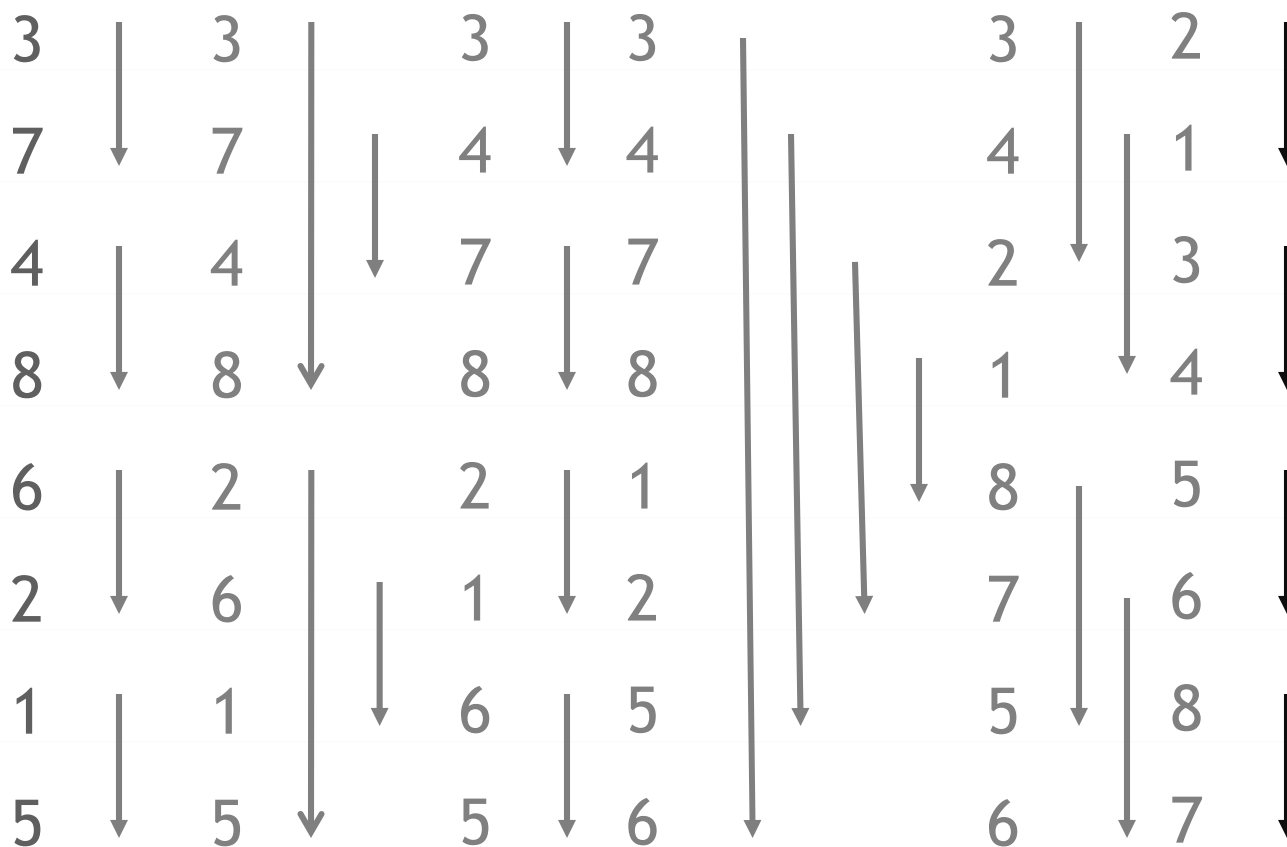
Subsequent steps have comparators in same direction.

# Bitonic Merge Sort (Stage 3, Step 2)



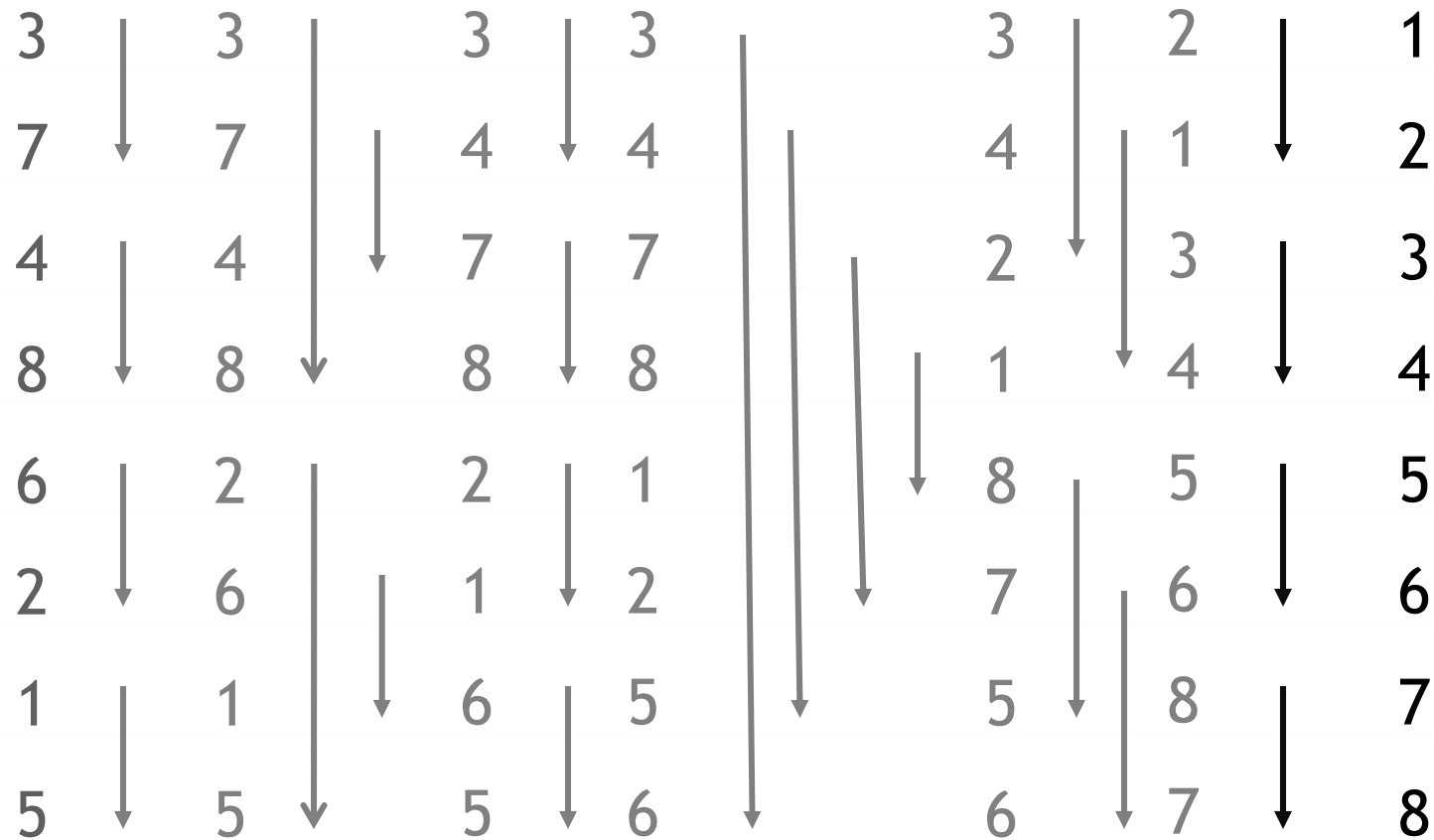
Subsequent steps have comparators in same direction.

# Bitonic Merge Sort (Stage 3, Step 1)



Subsequent steps have comparators in same direction.

# Bitonic Merge Sort



Sorted list of size 8.

# Bitonic Merge Sort Summary

---

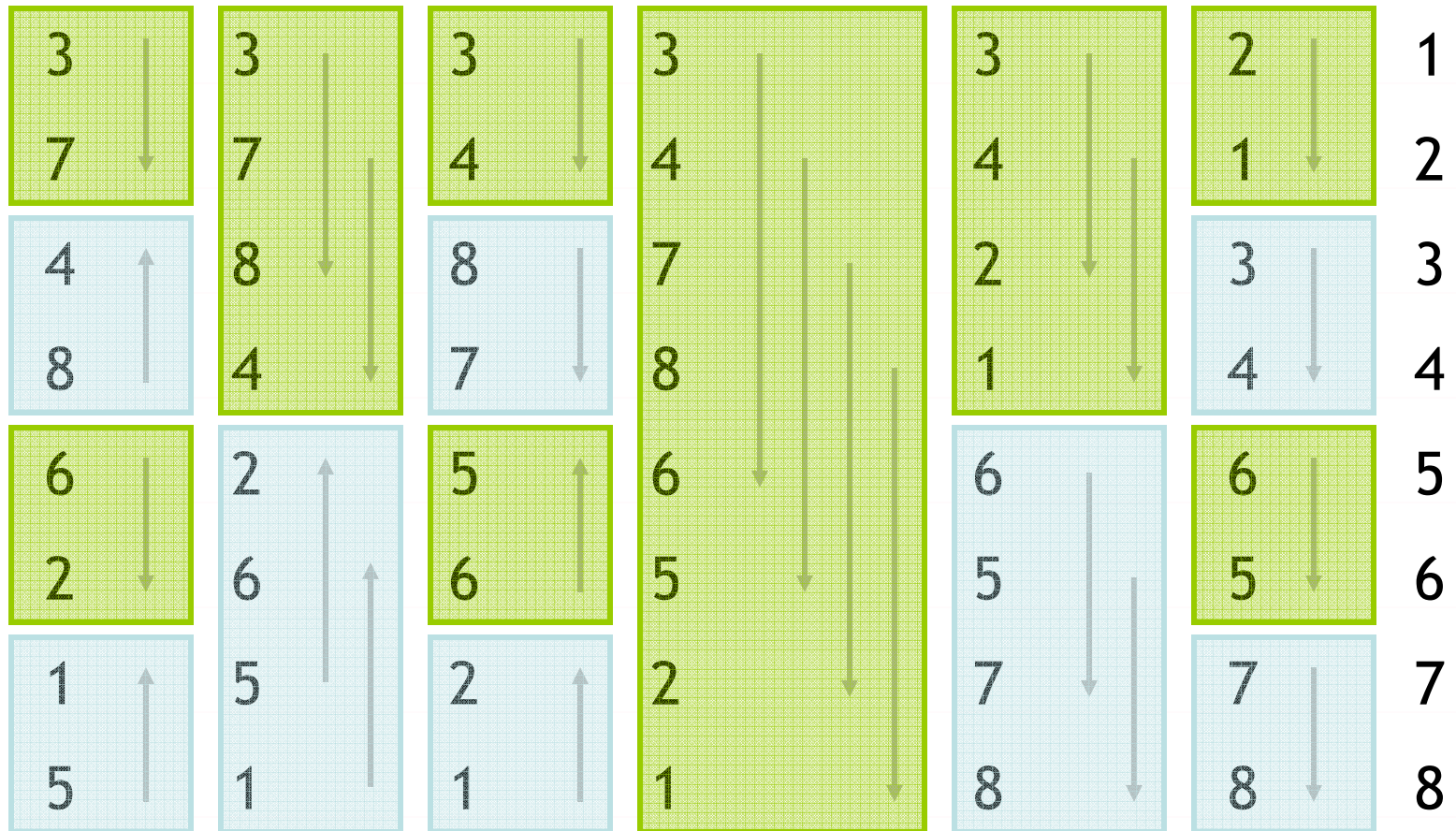
- **Separate rendering pass for each set of swaps**
  - $O(\log^2 n)$  passes
  - Each pass performs  $n$  compare/swaps
  - Total compare/swaps:  $O(n \log^2 n)$

# Making GPU Sorting Faster

---

- **Draw several quads with similar computation instead of single quad**
  - Reduce decision making in fragment program
- **Push work into vertex processor and interpolator**
  - Reduce computation in fragment program
- **More than one compare/swap per sort kernel invocation**
  - Reduce computational complexity

# Grouping Computation



# Implementation Details

---

- **Specify interpolants for smaller quads**
  - 'down' or 'up' compare and swap
  - distance to comparison partner
  
- **See Kipfer & Westermann article in GPU Gems 2, Kipfer et al. Graphics Hardware 04 and GPUABiSort for more details**

# GPU Sort

---

- Use blending operators for comparison
- Use texture mapping hw to map sorting op.

# 2D Memory Addressing

---

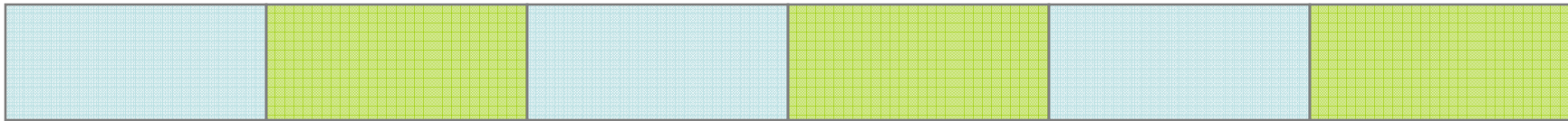
- **GPUs optimized for 2D representations**
  - Map 1D arrays to 2D arrays
  - Minimum and maximum regions mapped to row-aligned or column-aligned quads

# 1D - 2D Mapping

---

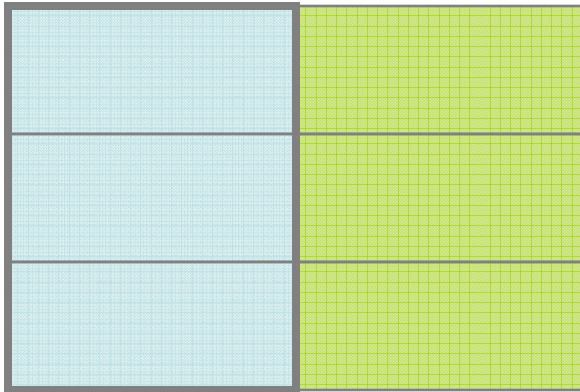
MIN

MAX



# 1D - 2D Mapping

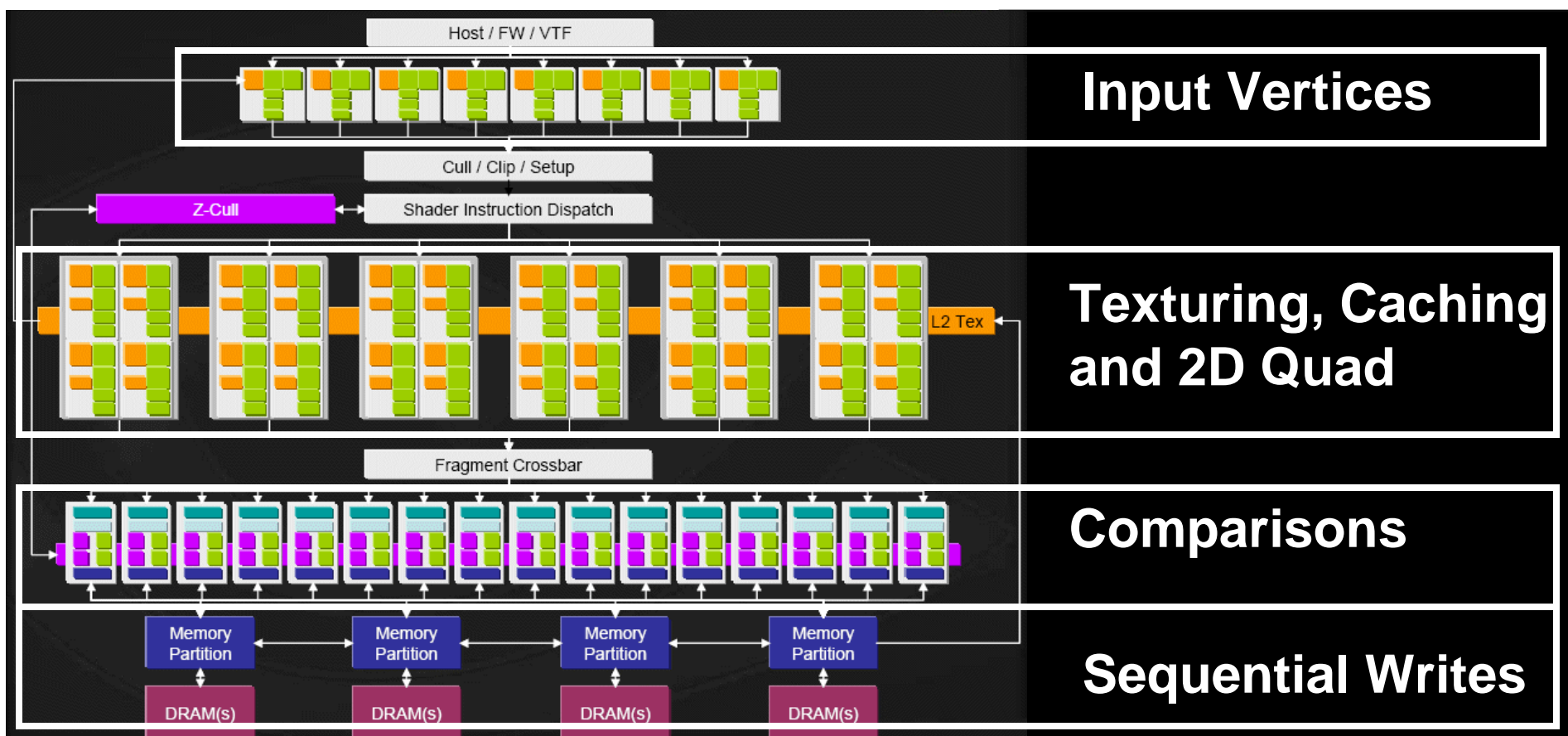
---



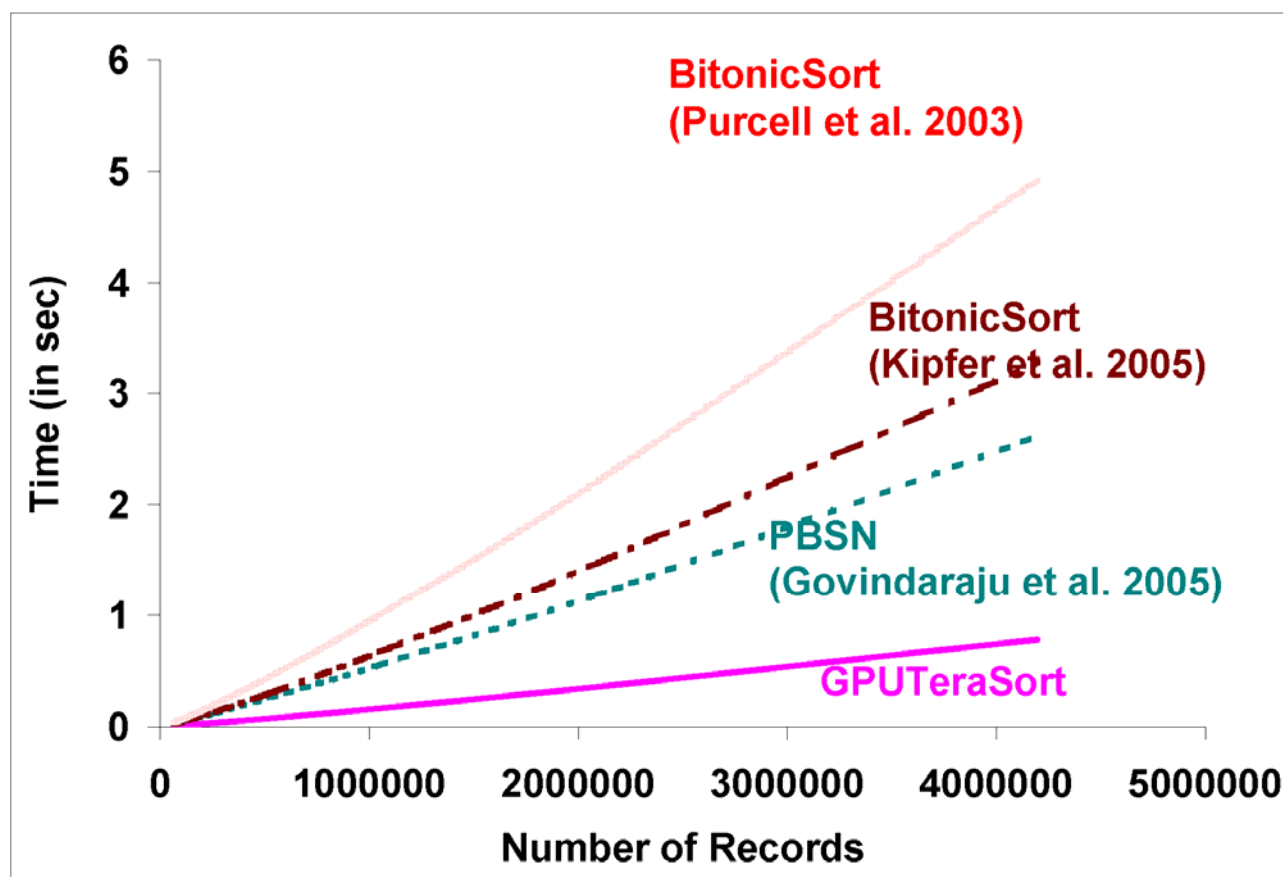
**MIN**

Effectively reduce instructions  
per element

# Sorting on GPU: Pipelining and Parallelism

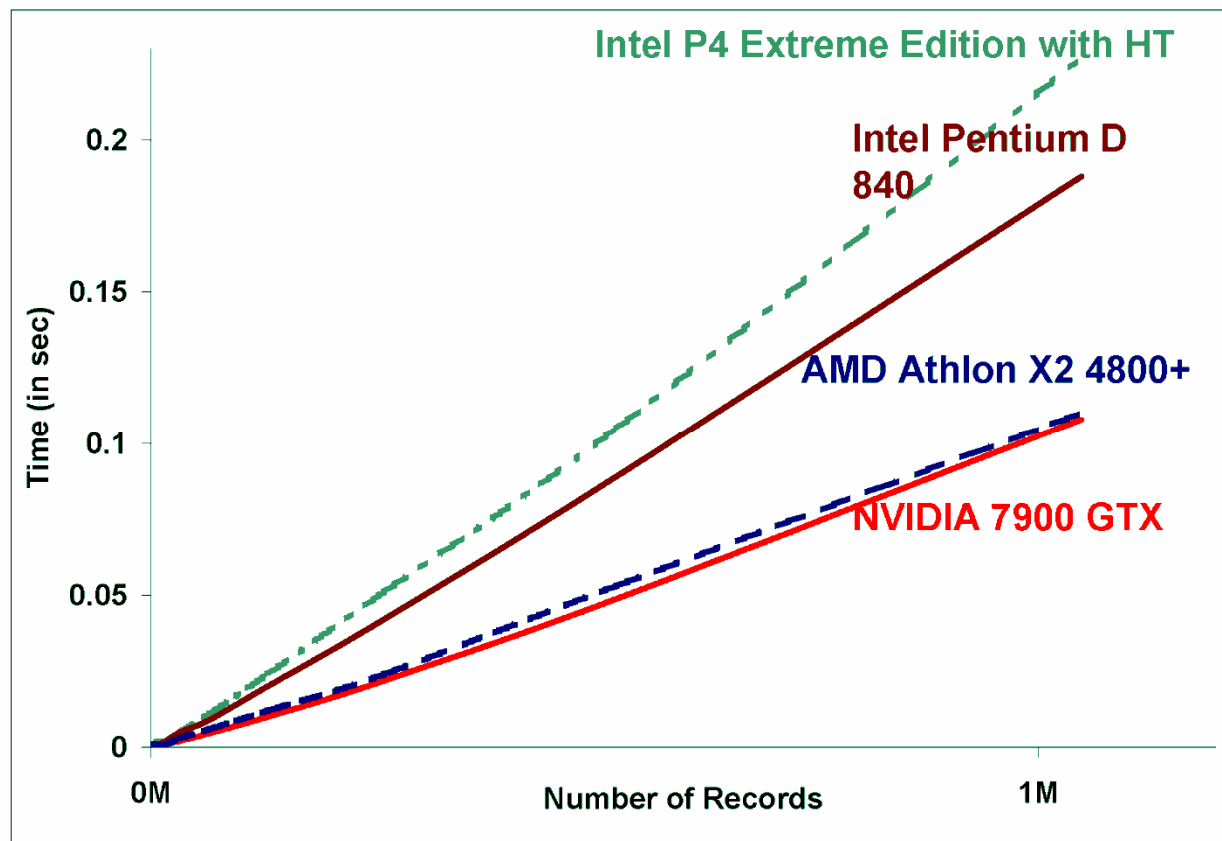


# Comparison with GPU-Based Algorithms



**3-6x faster than  
prior GPU-based  
algorithms!**

# GPU vs. High-End Multi-Core CPUs

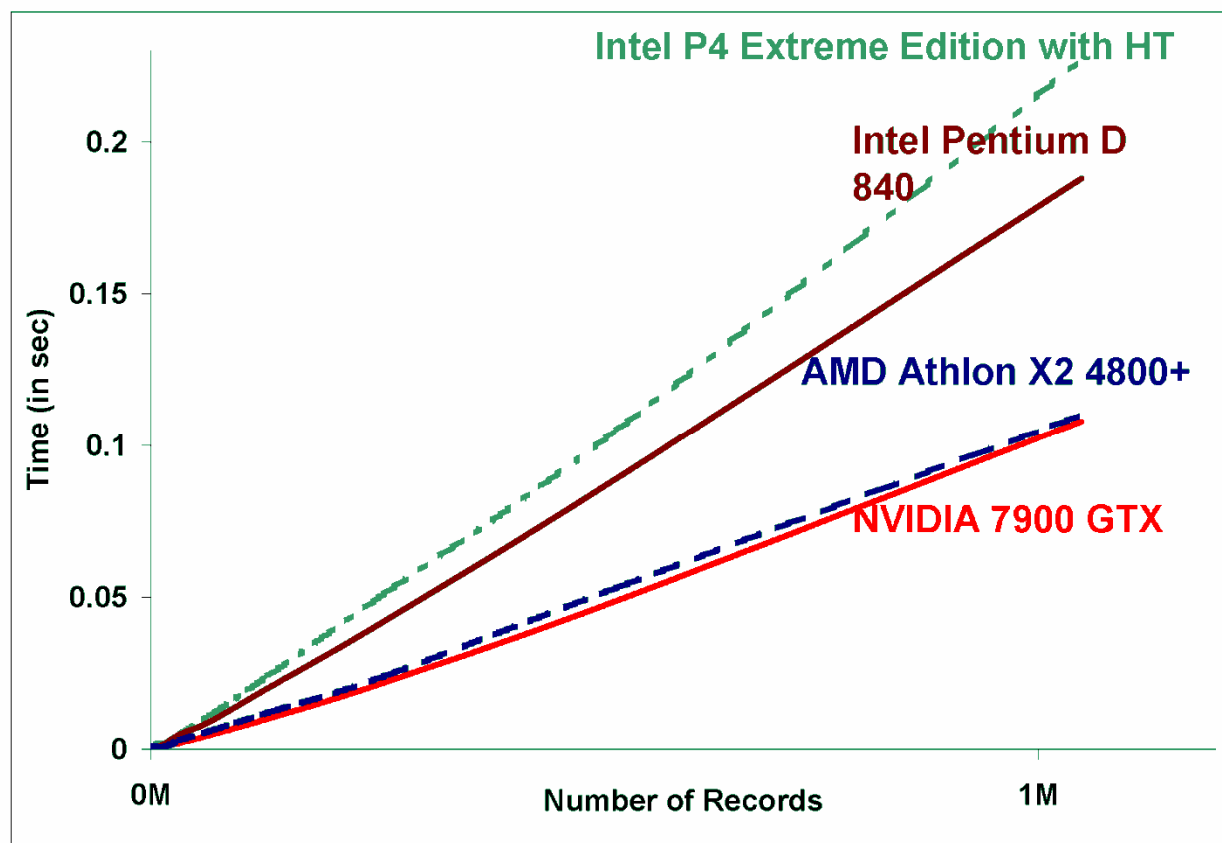


**2-2.5x faster than Intel Pentium D processors**

**Single GPU performance comparable to high-end dual core Athlon**

**Multi-threaded CPU code from Intel Corporation**

# GPU vs. High-End Multi-Core CPUs



**2-2.5x faster than Intel Pentium D processors**

**Single 7900 GPU performance comparable to high-end dual core Athlon**

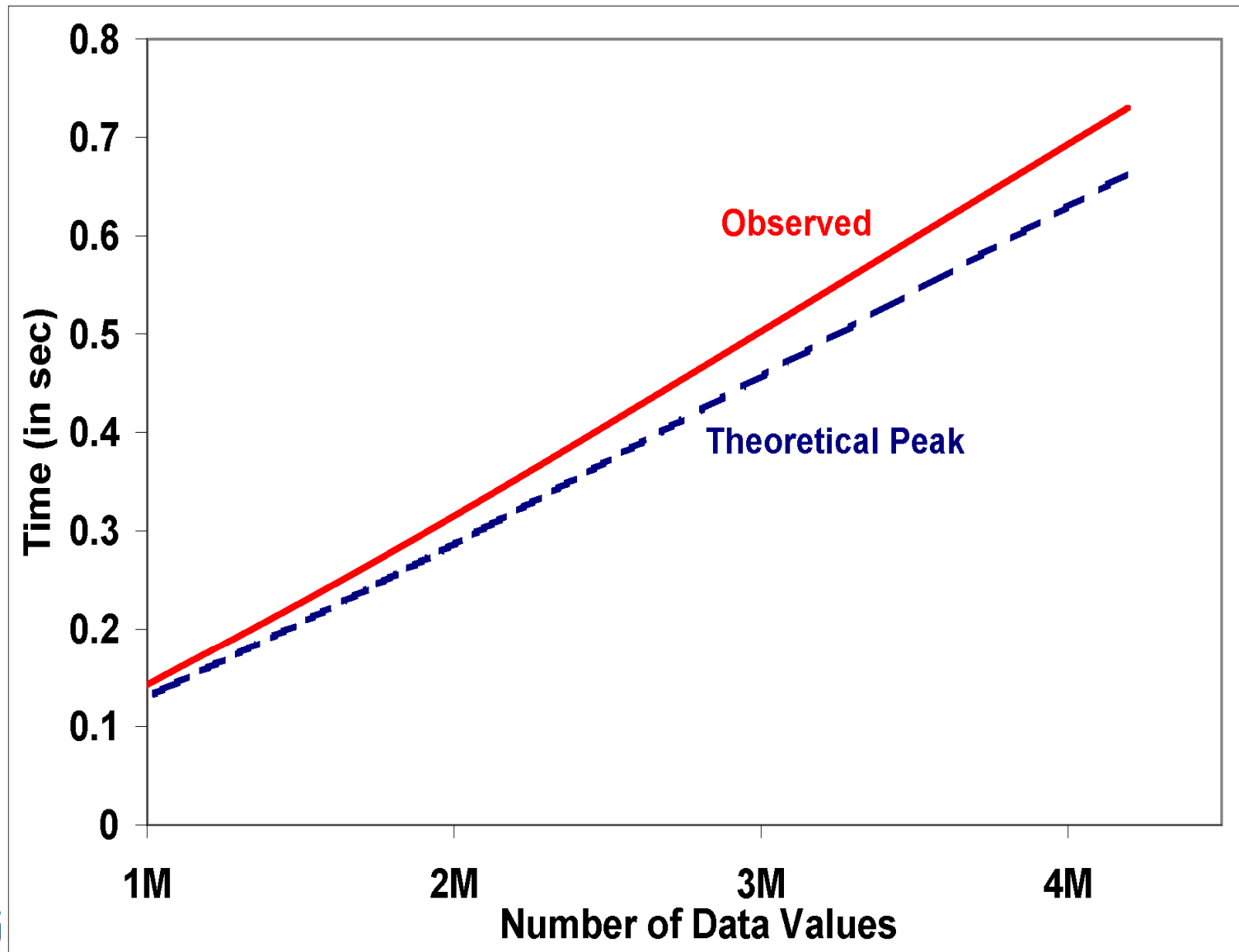
**Slash Dot and Toms Hardware Guide Headlines, June 2005**

# GPU Cache Model

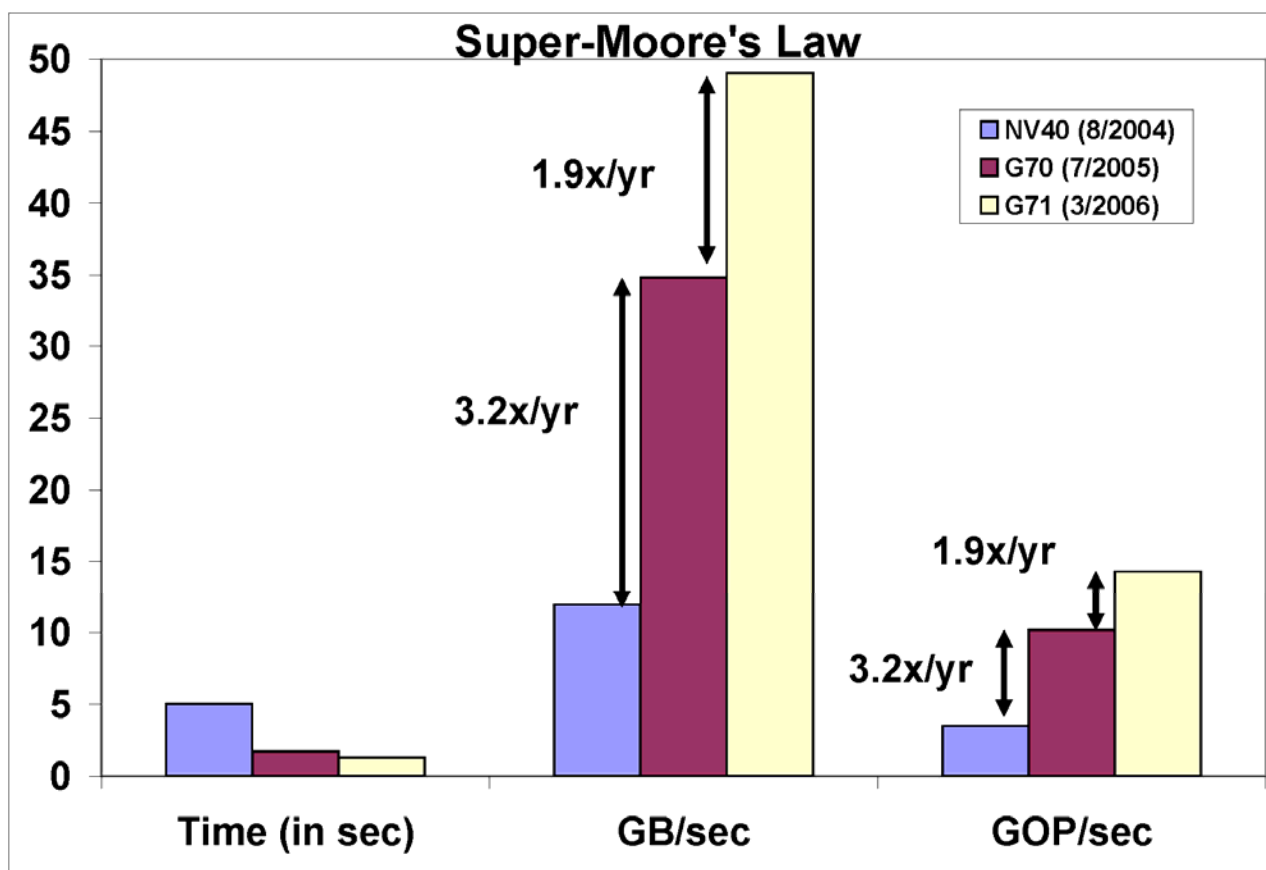
---

- **Small data caches**
  - Better hide the memory latency
  - Vendors do not disclose cache information - critical for scientific computing on GPUs
- **We design simple model**
  - Determine cache parameters (block and cache sizes)
  - Improve sorting, FFT and SGEMM performance

# Cache-Efficient Algorithm Performance



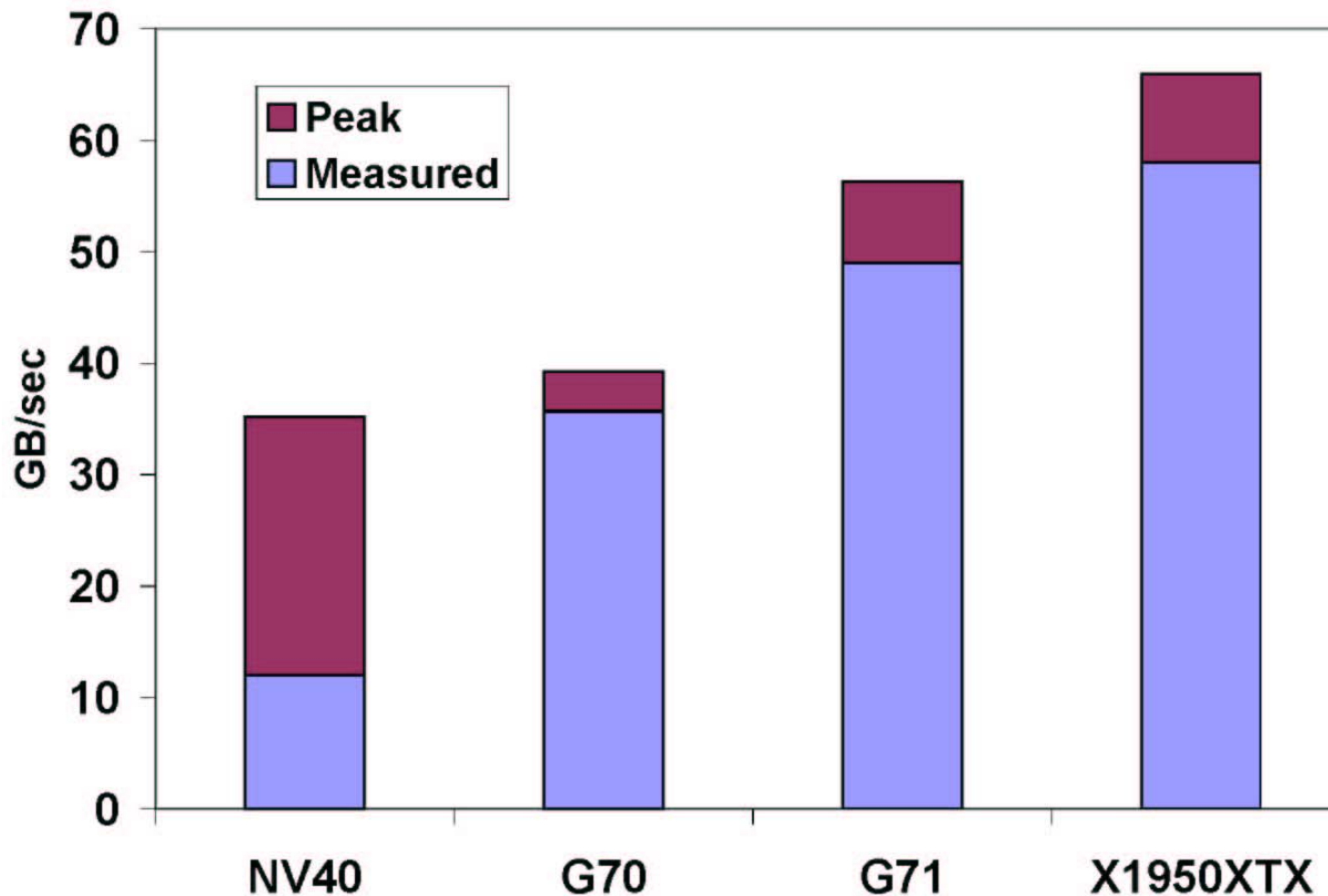
# Super-Moore's Law Growth



**50 GB/s** on a  
single GPU

**Peak Performance:**  
Effectively hide  
memory latency  
with **15 GOP/s**

# Memory Performance



# External Memory Sorting

---

- Performed on Terabyte-scale databases
- Two phases algorithm [Vitter01, Salzberg90, Nyberg94, Nyberg95]
  - Limited main memory
  - First phase - partitions input file into large data chunks and writes sorted chunks known as “Runs”
  - Second phase - Merge the “Runs” to generate the sorted file

# External Memory Sorting

---

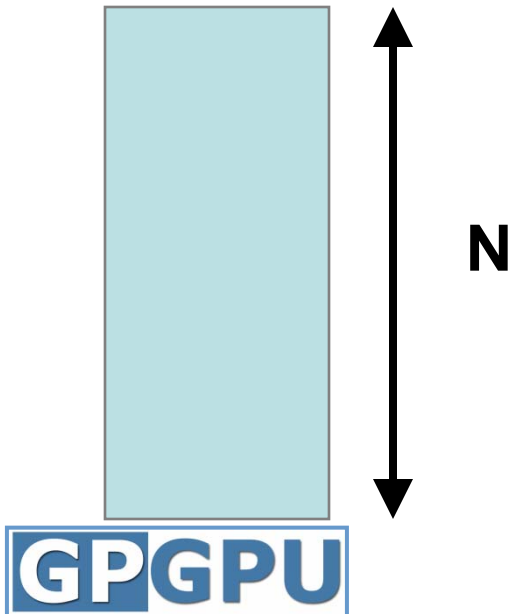
- Performance mainly governed by I/O

Salzberg Analysis: Given the main memory size  $M$  and the file size  $N$ , if the I/O read size per run is  $T$  in phase 2, external memory sorting achieves efficient I/O performance if the run size  $R$  in phase 1 is given by  $R \approx \sqrt{TN}$

# External Memory Sorting

---

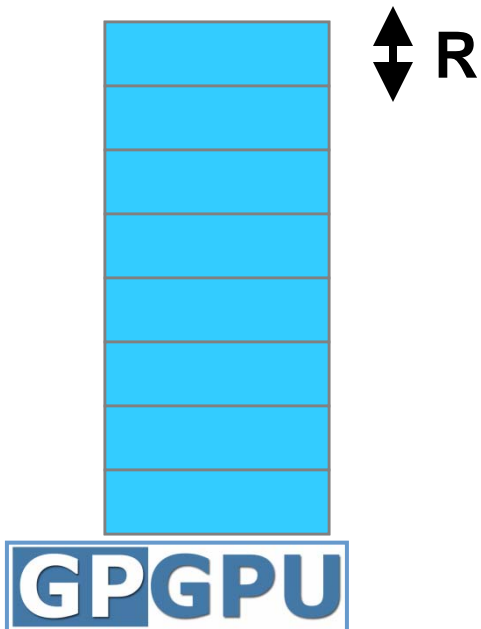
Given the main memory size  $M$  and the file size  $N$ , if the I/O read size per run is  $T$  in phase 2, external memory sorting achieves efficient I/O performance if the run size  $R$  in phase 1 is given by  $R \approx \sqrt{TN}$



# External Memory Sorting

---

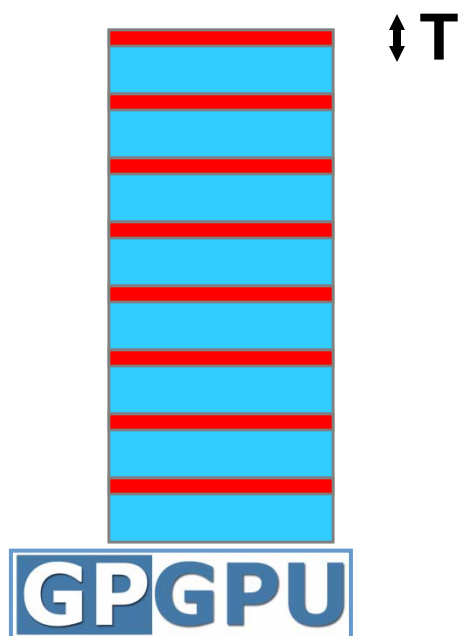
Given the main memory size  $M$  and the file size  $N$ , if the I/O read size per run is  $T$  in phase 2, external memory sorting achieves efficient I/O performance if the run size  $R$  in phase 1 is given by  $R \approx \sqrt{TN}$



# External Memory Sorting

---

Given the main memory size  $M$  and the file size  $N$ , if the I/O read size per run is  $T$  in phase 2, external memory sorting achieves efficient I/O performance if the run size  $R$  in phase 1 is given by  $R \approx \sqrt{TN}$



# Salzberg Analysis

---

- If  $N=100\text{GB}$ ,  $T=2\text{MB}$ , then  
 $R \approx 230\text{MB}$
- Large data sorting on CPUs can achieve high I/O performance by sorting large runs

# Massive Data Handling on CPUs

---

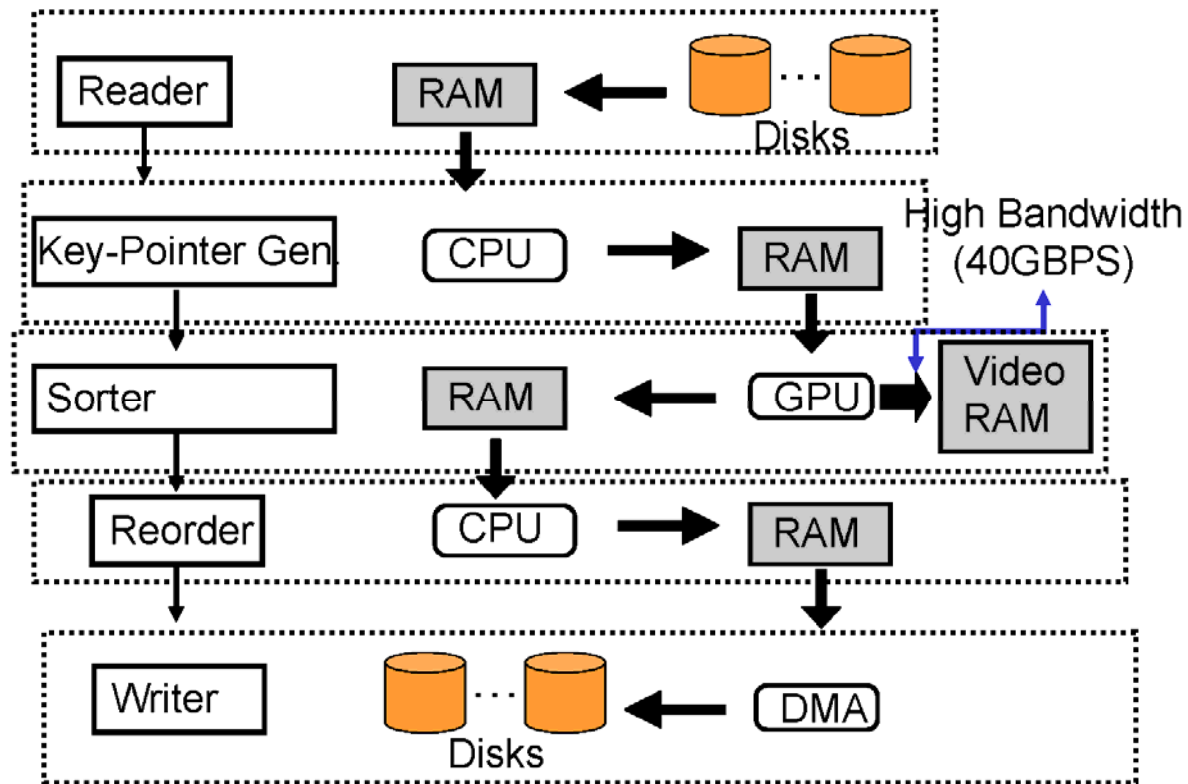
- **Require random memory accesses**
  - Small CPU caches (< 2MB)
  - Slower than even sequential disk accesses - bottleneck shift from I/O to memory
- **CPUs are deeply pipelined**
  - Do not hide latency - high cycles per instruction (CPI)
  - Huge memory to compute gap!
- **CPU is under-utilized for data intensive applications**

# External memory sorting

---

- **External memory sorting on CPUs can have low performance due to**
  - High memory latency on account of cache misses
  - Or low I/O performance
  
- **Sorting is hard!**

# GPURTeraSort



# Implementation & Results

---

- Pentium IV CPU (\$170)
  - NVIDIA 7800 GT (\$270)
  - 2 GB RAM (\$152)
  - 9 80GB SATA disks (\$477)
  - SuperMicro Motherboard & SATA Controller (\$325)
  - Windows XP
- 
- PC costs \$1469 in April 06 - now <\$1K

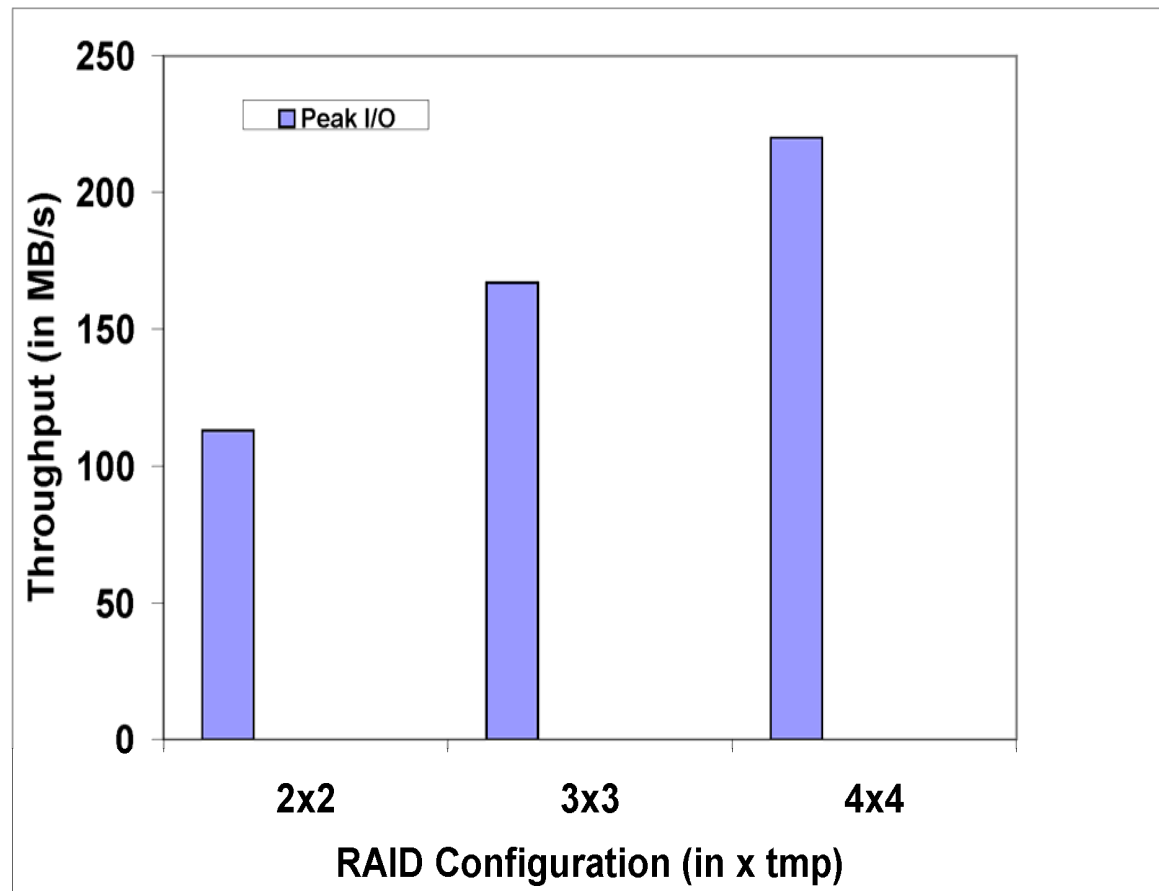
# Implementation & Results

---

- **Indy SortBenchmark**
  - 10 byte random string keys
  - 100 byte long records
  - Sort maximum amount in 644 seconds

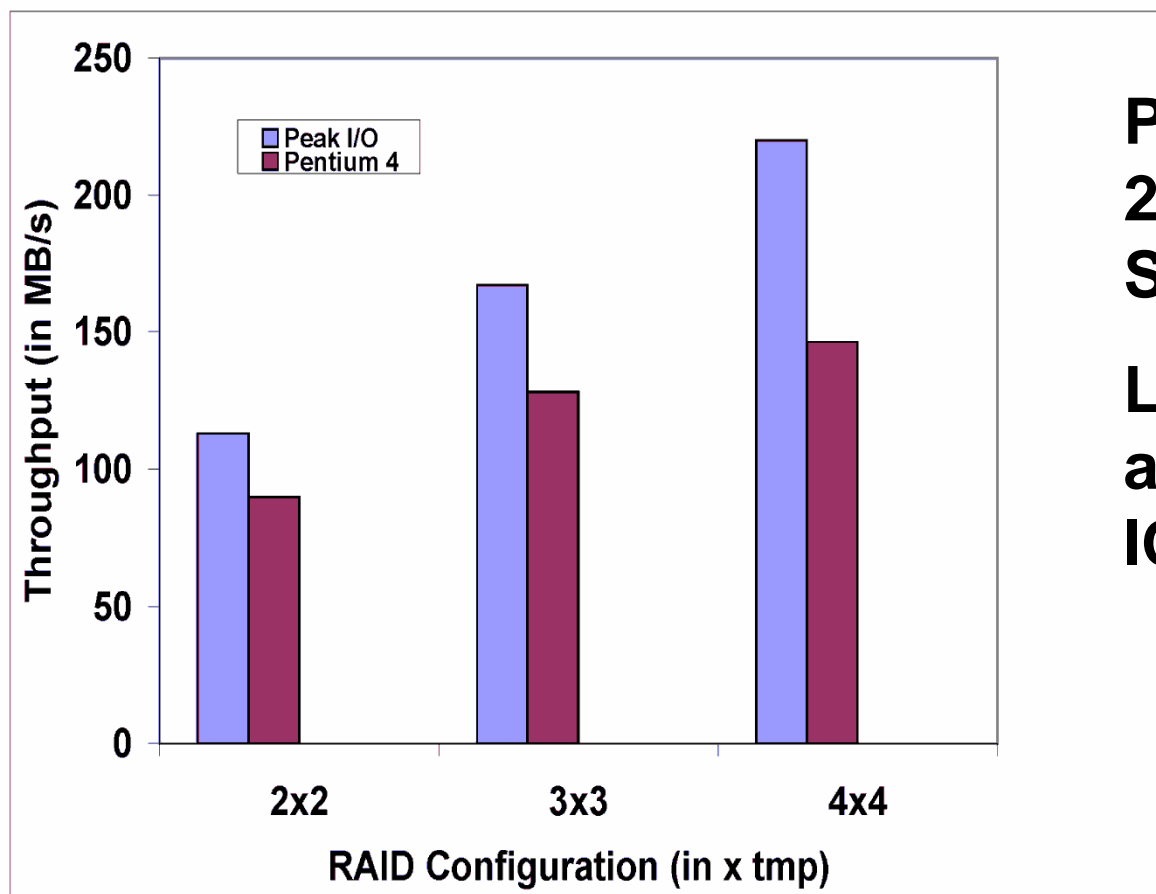
# I/O Performance

## Salzberg Analysis: 100 MB Run Size



# I/O Performance

## Salzberg Analysis: 100 MB Run Size

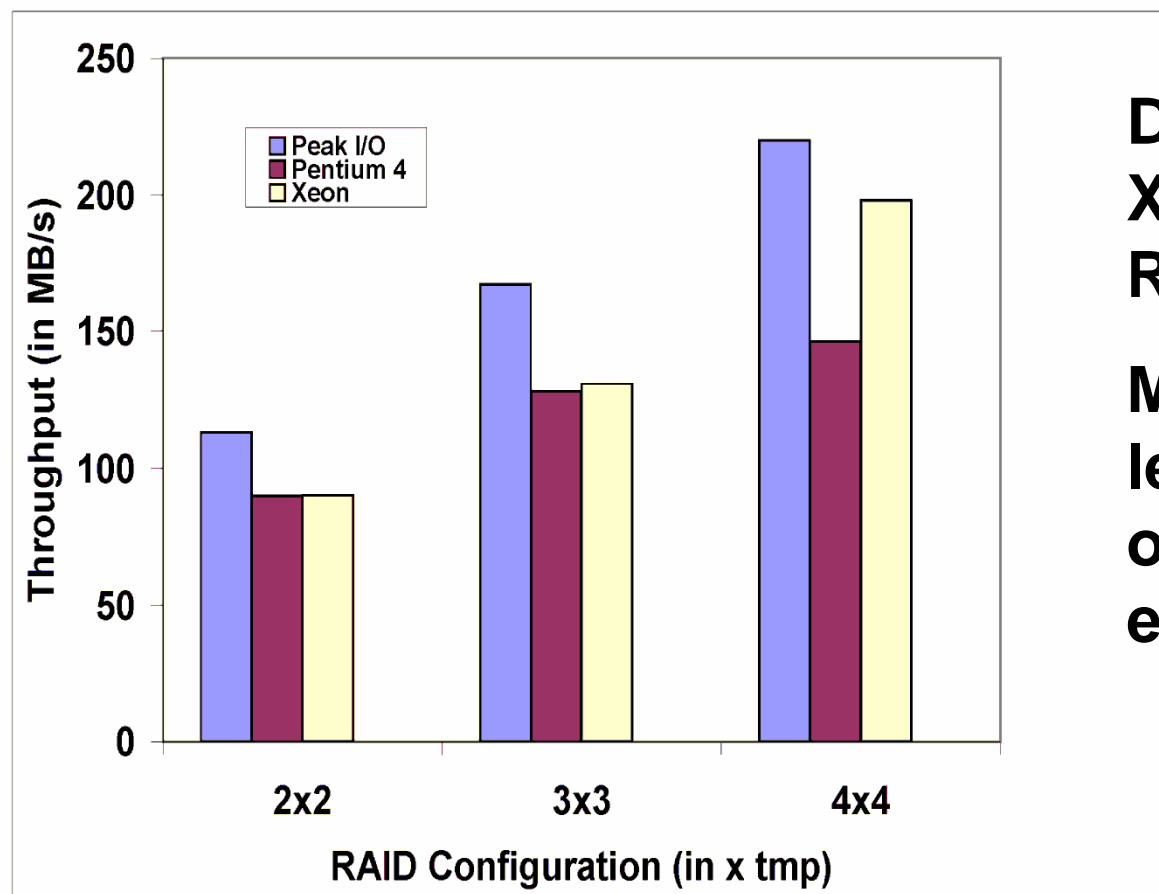


**Pentium IV:  
25MB Run  
Size**

**Less work  
and only 75%  
IO efficient!**

# I/O Performance

## Salzberg Analysis: 100 MB Run Size

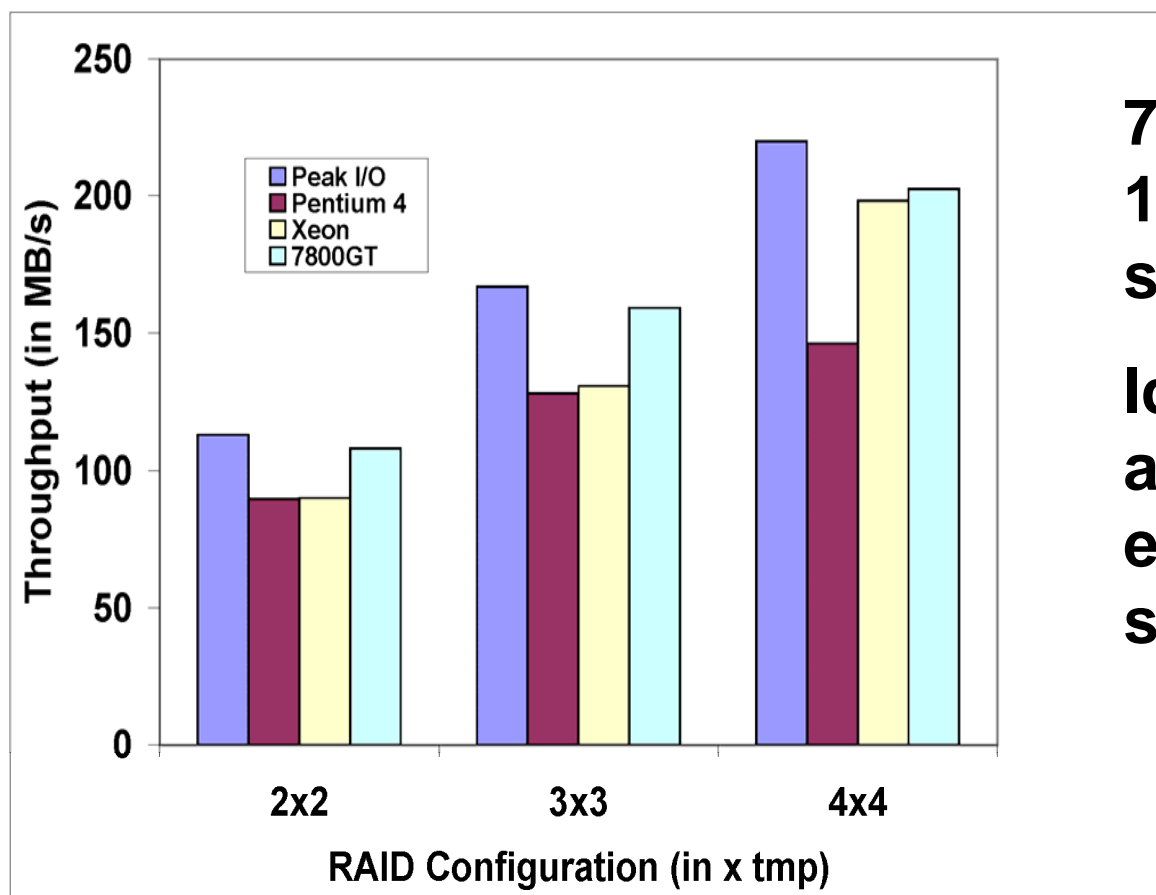


**Dual 3.6 GHz  
Xeons: 25MB  
Run size**

**More cores,  
less work but  
only 85% IO  
efficient!**

# I/O Performance

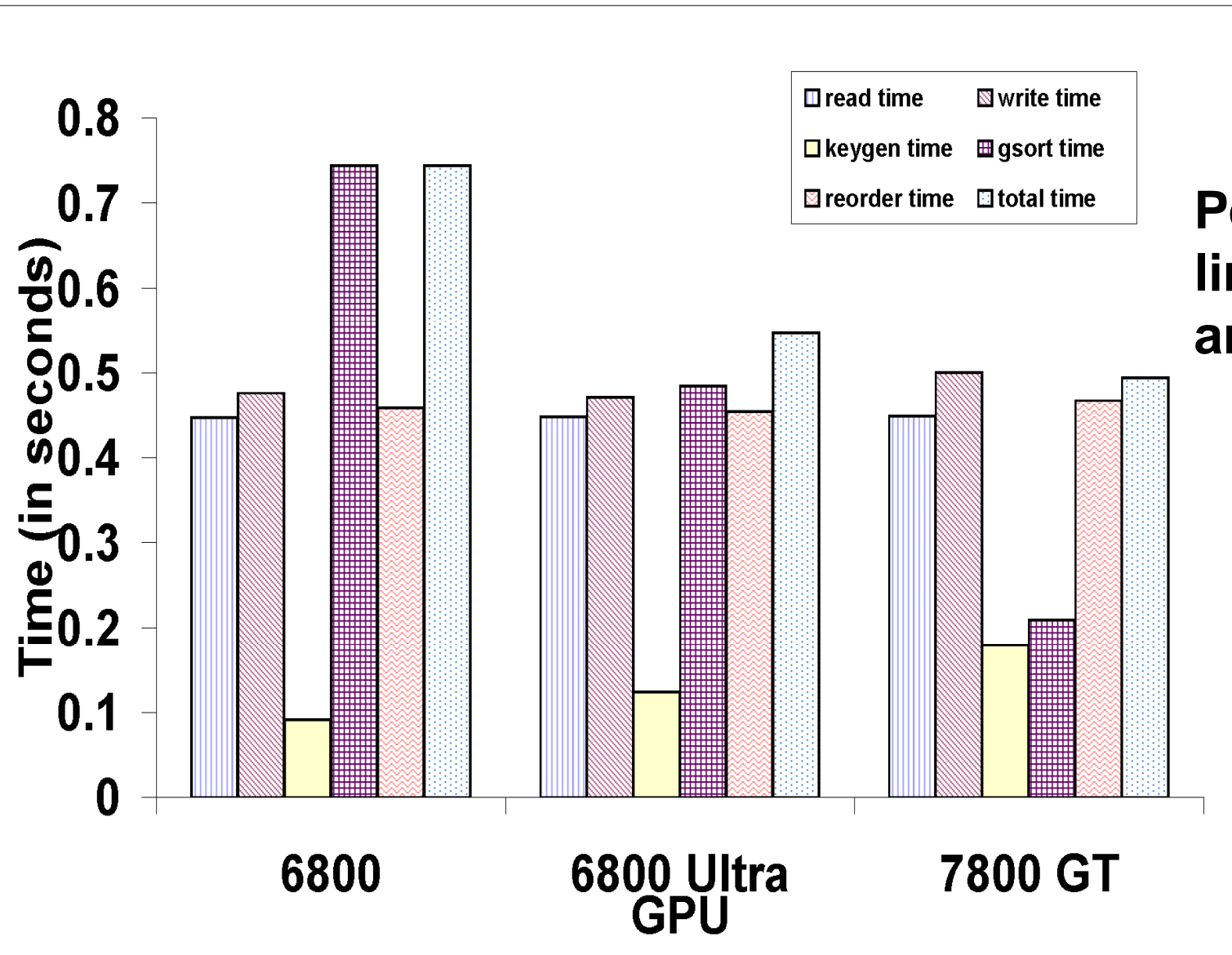
## Salzberg Analysis: 100 MB Run Size



**7800 GT:  
100MB run  
size**

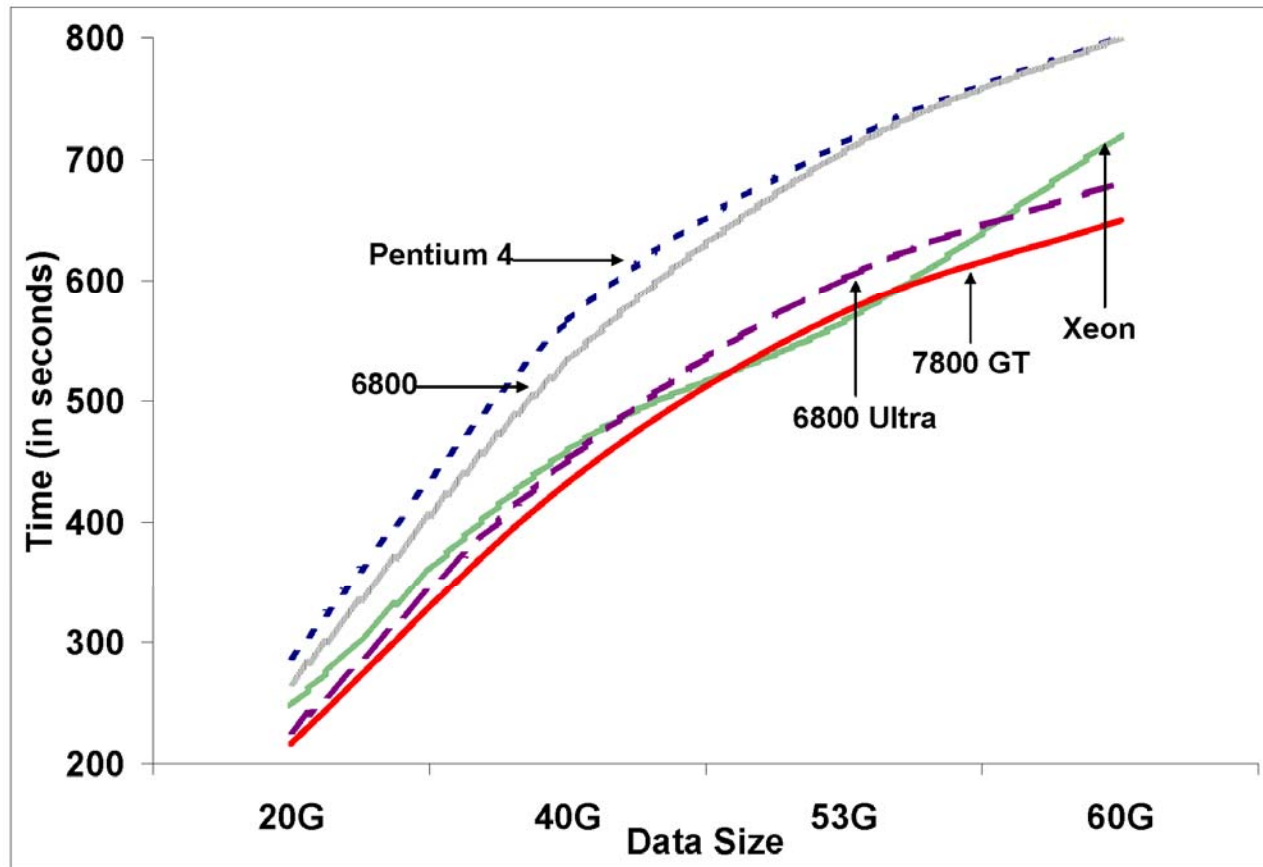
**Ideal work,  
and 92% IO  
efficient with  
single CPU!**

# Task Parallelism



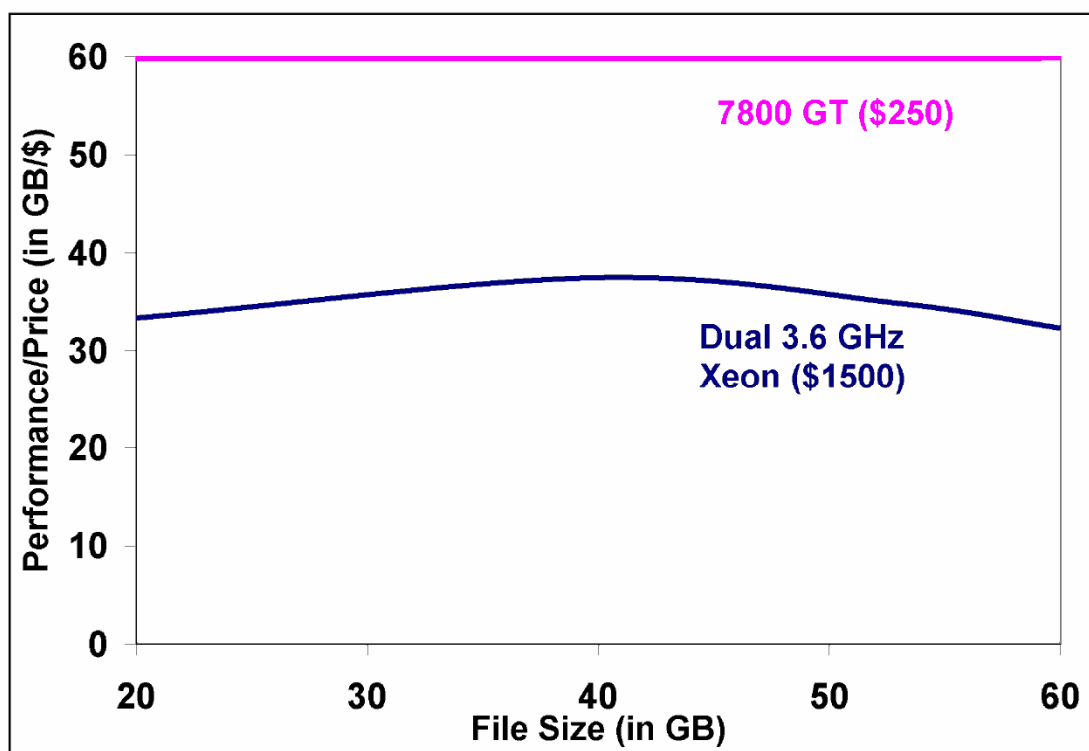
**Performance limited by IO and memory**

# Overall Performance



**Faster and more scalable than Dual Xeon processors (3.6 GHz)!**

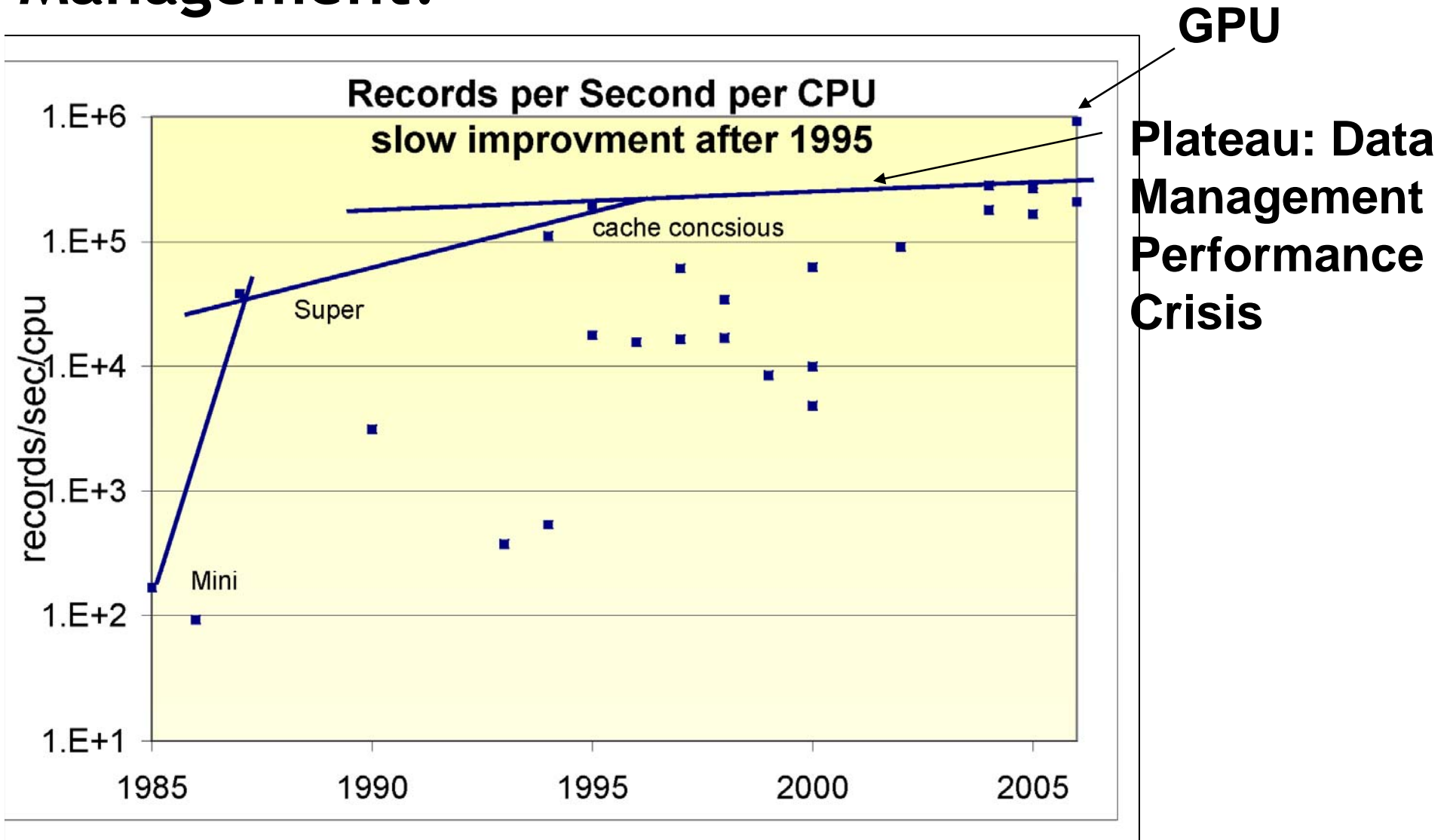
# Performance/\$



**1.8x faster than  
current Terabyte  
sorter**

**World's best  
performance/\$  
system**

# Why GPU-like Architectures for Large Data Management?



# Advantages

---

- **Exploit high memory bandwidth on GPUs**
  - Higher memory performance than CPU-based algorithms
- **High I/O performance due to large run sizes**

# Advantages

---

- **Offload work from CPUs**
  - CPU cycles well-utilized for resource management
- **Scalable solution for large databases**
- **Best performance/price solution for terabyte sorting**

# Searching for Quantiles

---

- Given a set of values in the GPU, compute the **Kth-largest** number
- Traditional CPU algorithms require arbitrary data writes - require new algorithm without
  - Data rearrangement
  - Data readback to CPU
- Our solution - search for the **Kth-largest** number

# K-th Largest Number

---

- Let  $v_k$  denote the k-th largest number
- How do we generate a number  $m$  equal to  $v_k$ ?
  - Without knowing  $v_k$ 's value
  - Count the number of values  $\geq$  some given value
  - Starting from the most significant bit, determine the value of each bit at a time

# K-th Largest Number

---

- **Given a set  $S$  of values**
  - $c(m)$  — number of values  $\geq m$
  - $v_k$  — the  $k$ -th largest number
- **We have**
  - If  $c(m) \geq k$ , then  $m \leq v_k$
  - If  $c(m) < k$ , then  $m > v_k$
- **$c(m)$  computed using occlusion queries**

# 2<sup>nd</sup> Largest in 9 Values

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 0000$   
 $v_2 = 1011$

# Draw a Quad at Depth 8 Compute $c(1000)$

---



0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1000$   
 $v_2 = 1011$

# 1<sup>st</sup> bit = 1

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1000$

$v_2 = 1011$

$c(m) = 3$

# Draw a Quad at Depth 12 Compute c(1100)

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1100$   
 $v_2 = 1011$

# 2<sup>nd</sup> bit = 0

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$$m = 1100$$

$$v_2 = 1011$$

$$c(m) = 1$$

# Draw a Quad at Depth 10 Compute c(1010)

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1010$   
 $v_2 = 1011$

# 3<sup>rd</sup> bit = 1

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$$m = 1010$$
$$v_2 = 1011$$

$$c(m) = 3$$

# Draw a Quad at Depth 11 Compute c(1011)

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1011$   
 $v_2 = 1011$

# 4<sup>th</sup> bit = 1

---

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1011$   
 $v_2 = 1011$

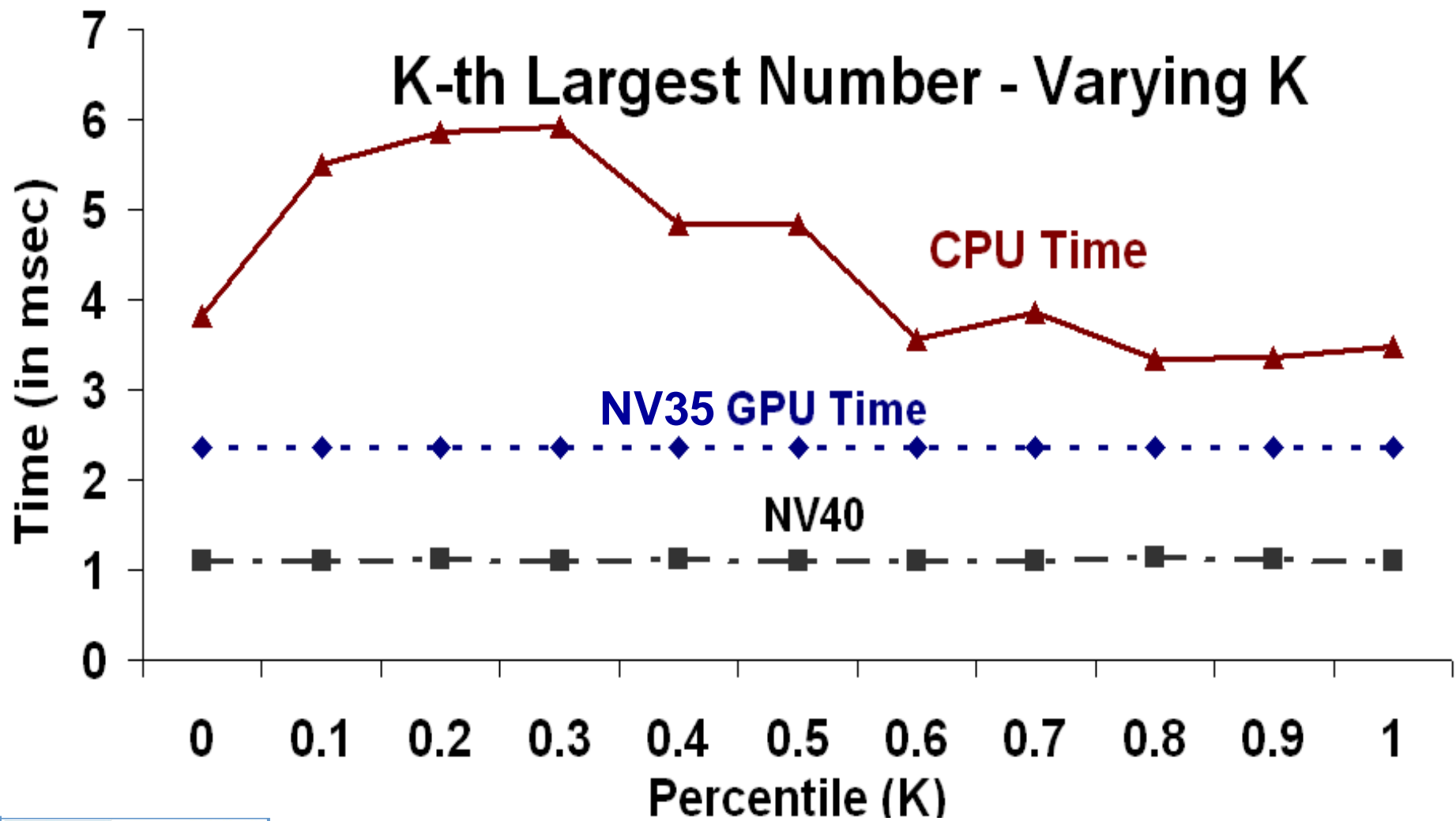
$c(m) = 2$

# Our algorithm

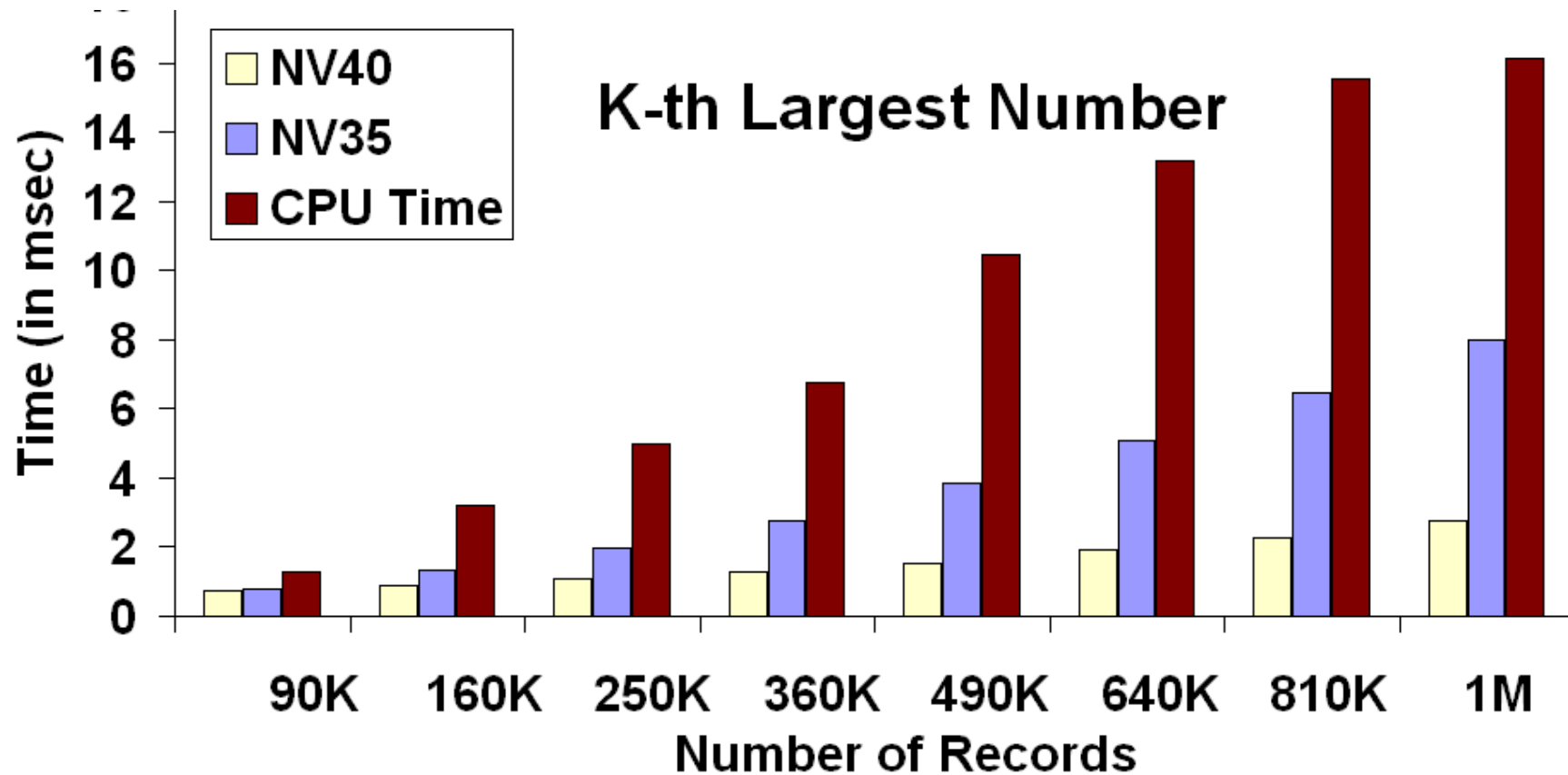
---

- Initialize  $m$  to 0
- Start with the *MSB* and scan all bits till *LSB*
- At each bit, put 1 in the corresponding bit-position of  $m$
- If  $c(m) < k$ , make that bit 0
- Proceed to the next bit

# Kth-Largest



# Median



**3x performance improvement per year!**