SortBenchmark: A Benchmark for Many-core Computing

Naga K. Govindaraju
Microsoft Many-core Incubation
Sorting

“I believe that virtually every important aspect of programming arises somewhere in the context of sorting or searching!”
Don Knuth, Stanford University

There is a lot of effort on multi-core processors, and comparatively little effort on addressing the “core” problems: (1) the memory architecture, and (2) the way processors access memory. Sort demonstrates those problems very clearly.
Jim Gray, Microsoft Research
Sorting

- Well studied
  - High performance computing
  - Databases
  - Computer graphics
  - Programming languages
  - ...
- Google map reduce algorithm
- Spec benchmark routine!
Massive Databases

Terabyte-data sets are common
- Google sorts more than 100 billion terms in its index
- > 1 Trillion records in web indexed!

Database sizes are rapidly increasing!
- Max DB sizes increases 3x per year (http://www.wintercorp.com)
- Processor improvements not matching information explosion
CPU vs. GPU

CPU
(3 GHz)
4 x 6 MB Cache

System Memory
(16 GB)

GPU (690 MHz)
Video Memory
(4 GB)

GPU (690 MHz)
Video Memory
(4 GB)

GPU (690 MHz)
Video Memory
(4 GB)

PCI-E Bus
(8 GB/s)
Massive Data Handling on CPUs

- Require random memory accesses
  - Small CPU caches
  - Random memory accesses slower than even sequential disk accesses

- High memory latency
  - Huge memory to compute gap!

- CPUs are deeply pipelined
  - Pentium 4 has 30 pipeline stages
  - Do not hide latency - high cycles per instruction (CPI)

- CPU is under-utilized for data intensive applications
Why Many-Core Sort?

http://research.microsoft.com/barc/SortBenchmark

- Records per Second per CPU
- slow improvement after 1995
- cache conscious
- Super
- Mini
- GPU TeraSort

1. Records per Second per CPU
2. Graph showing records per second per CPU from 1985 to 2005
3. Mini, Super, and cache conscious categories on the graph
4. GPU TeraSort highlighted with an arrow
Massive Data Handling on CPUs

- Sorting is hard!

- GPU a potentially scalable solution to terabyte sorting and scientific computing
  - We provide a scalable solution on GPUs
Graphics Processing Units (GPUs)

- *Commodity processor for graphics applications*
- Massively data-parallel processors
- High memory bandwidth
  - Low memory latency pipeline
  - Programmable
- High growth rate
GPU: Commodity Processor

- Cell phones
- Laptops
- Consoles
- Desktops
- PSP
Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- *Massively data-parallel processors*
  - 10x more operations per sec than CPUs
- High memory bandwidth
  - Better hides memory latency pipeline
  - Programmable
- High growth rate
Parallelism on GPUs

Peak FLOPS
GPU – 933 GFLOPS
CPU – 100 GFLOPS
Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- Massively data-parallel processors
- **High memory bandwidth**
  - Better hides latency pipeline
  - Programmable
  - **10x more memory bandwidth than CPUs**
- High growth rate
Traditional GPGPU Pipeline

Very high parallelism

Input Assembler
Vertex Shader
Tessellation
Geometry Shader
Rasterizer
Pixel Shader
Output Merger

140 GB/s

Memory

Hides memory latency!!
DirectX11: Compute Shader
Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- Massively data-parallel processors
- High memory bandwidth
  - Better hides latency pipeline
  - Programmable
- *High growth rate*
Memory Performance on GPUs

![Graph showing memory performance on GPUs from February 2002 to February 2008. The x-axis represents the years with increments of one year, and the y-axis represents GB/s with increments of 20 GB/s.]
**GPUs for Sorting: Issues**

- Random writes are expensive
  - Optimized CPU algorithms do not map!
- Lack of support for recursion
- Out-of-core algorithms
  - Limited GPU memory
Outline

- Overview
- Sorting on GPUs
- Conclusions and Future Directions
Sorting on GPUs

- Adaptive sorting algorithms
  - Extent of sorted order in a sequence

- General sorting algorithms

- External memory sorting algorithms
Adaptive Sorting on GPUs

- Prior adaptive sorting algorithms require random data writes
  - In insertion sort, a processor may operate on different number of elements – load imbalance
- GPUs optimized for data-parallel algorithms
  - Use optimized data-parallel primitives such as scans
- Design adaptive sorting using only data-parallel computations
  - Avoid load imbalance by operating on same number of elements on each processor
Adaptive Sorting Algorithm

- Multiple iterations

- Each iteration uses a two pass algorithm
  - First pass – Compute an increasing sequence $M$
  - Second pass - Compute the sorted elements in $M$
  - Iterate on the remaining unsorted elements

Increasing Sequence

Given a sequence $S=\{x_1,..., x_n\}$, an element $x_i$ belongs to $M$ if and only if $x_i \leq x_j$, $i<j$, $x_j$ in $S$. 
Increasing Sequence

M is an increasing sequence
Increasing Sequence Computation

\[ X_1 \ X_2 \ \ldots \ X_{i-1} \ X_i \ X_{i+1} \ \ldots \ X_{n-1} \ X_n \]
Increasing Sequence Computation

\[ X_n \leq \infty \]
Increasing Sequence Computation

Yes.
Prepend $x_i$ to $M$
$Min = x_i$

$x_i \leq Min$?

$x_i \quad x_{i+1} \quad \ldots \quad x_{n-1} \quad x_n$

Compute
Increasing Sequence Computation

\[ x_1 \leq \{x_2, \ldots, x_n\} \]

Compute

\[ X_1 \quad X_2 \quad \ldots \quad X_{i-1} \quad X_i \quad X_{i+1} \quad \ldots \quad X_{n-1} \quad X_n \]
Computing Sorted Elements

**Theorem 1:** Given the increasing sequence $M$, rank of an element $x_i$ in $M$ is determined if $x_i < \min (I-M)$
Computing Sorted Elements

\[ X_1, X_2, X_3, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_{n-2}, X_{n-1}, X_n \]
Computing Sorted Elements

\[
\begin{align*}
X_1 & \geq X_2 \geq X_3 \ldots \geq X_{i-1} \\
x_i & \leq X_{i+1} \leq X_{n-2} \leq X_{n-1} \leq X_n
\end{align*}
\]
Computing Sorted Elements

- Linear-time algorithm
  - Maintaining minimum
Computing Sorted Elements

\[ X_1 \quad X_2 \quad X_3 \cdots \quad X_{i-1} \quad X_i \quad X_{i+1} \cdots \quad X_{n-2} \quad X_{n-1} \quad X_n \]
Computing Sorted Elements

Compute

\[ X_1 \ X_2 \ \ldots \ X_{i-1} \ X_i \]

\( X_i \text{ in } M? \)

\( X_i \leq \text{min?} \)

\( X_i \text{ in } M? \)

\( X_i \leq \text{min?} \)

No. Update min

Yes. Append \( X_i \) to sorted list
Computing Sorted Elements

Compute

\[ X_1 \quad X_2 \quad \ldots \quad X_{i-1} \quad X_i \quad X_{i+1} \quad \ldots \quad X_{n-1} \quad X_n \]
Algorithm Analysis

Knuth’s measure of disorder:
Given a sequence $I$ and its longest increasing sequence $\text{LIS}(I)$, the sequence of disordered elements $Y = I - \text{LIS}(I)$

Theorem 2: *Given a sequence $I$ and $\text{LIS}(I)$, our adaptive algorithm sorts in at most $(2 \|Y\| + 1)$ iterations*
Pictorial Proof

\[ X_1 \ X_2 \ \ldots X_l \ X_{l+1} \ X_{l+2} \ \ldots X_m \ X_{m+1} \ X_{m+2} \ \ldots X_q \ X_{q+1} \ X_{q+2} \ \ldots X_n \]
Pictorial Proof

2 iterations 2 iterations 2 iterations

$X_1 \ X_2 \ \ldots X_l \ X_{l+1}$

$X_{l+2} \ \ldots X_m \ X_{m+1}$

$X_{m+2} \ \ldots X_q \ X_{q+1}$

$X_{q+2} \ \ldots X_n$
Example

1 2 8 3 4 5 6 7 9

V
Example

<table>
<thead>
<tr>
<th>Sorted</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scan: All-prefix-sums

For an input sequence \( A = [a_0, a_1, \ldots, a_{n-1}] \) and binary associative operation \( \oplus \) with left identity \( \varepsilon_\oplus \)

\[
\text{scan}_\oplus(A) = [\varepsilon_\oplus, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2}]
\]

Example: \( \oplus \) is addition, \( \varepsilon_\oplus = 0 \)

\[
A = [1, 1, 1, 1]
\]

\[
\text{scan}(A) = [0, 1, 2, 3]
\]
Computing Increasing Sequence

- Compute MIN scan in backward direction: $I'$
- Compare each element in $I$ with $I'$
  - Elements less than corresponding elements in $I'$ are in $M$
Computing Sorted Elements

Compute forward MIN scan of elements in \( I - M: I'' \)

- Elements not in \( M \) are set to \( \infty \) in \( I \) and scan is performed on \( I \)
- Compare elements in \( M \) with \( I \)
- Elements that are less than or equal to corresponding elements in \( I \) are sorted
Data Parallel Scans

- Well-studied in parallel computing
  - Implemented on GPUs [Horn 05, Harris et al. 07, Sengupta et al. 07, Dotsenko et al. 08]
  - Exploit shared memory to achieve higher memory efficiency

- Optimized libraries available

- Tree-based scan algorithms may not scale well with shared memory
  - Instead use matrix-based representations
Forward Unsegmented Scan

Sequence length, 4-byte words

Relative elapsed time

CUDPP-gems3.2
CUDPP-1.0alpha
Ours

Lower is better
Adaptive Sorting: Lessons

- Do not try to remap *serial* adaptive algorithms
- Design better data parallel algorithms to achieve scalable performance
  - Linear in the input size and sorted extent
  - Works well on almost sorted input
- Use data parallel primitives such as scans
General Sorting on GPUs

- General datasets
- High performance
General Sorting on GPUs

- Design sorting algorithms with deterministic memory accesses
  - Achieve high memory bandwidth
  - Can better hide the memory latency!!

- Require minimum and maximum computations
  - Low branching overhead

- No data dependencies
  - Utilize high parallelism on GPUs
GPU-Based Sorting Networks

- Represent data as 2D arrays

- Multi-stage algorithm
  - Each stage involves multiple steps

- In each step
  1. Compare one array element against exactly one other element at fixed distance
  2. Perform a conditional assignment (MIN or MAX) at each element location
Sorting Animation
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
1D – 2D Mapping

Effectively reduce instructions per element
Sorting on GPU: Pipelining and Parallelism

- Input Vertices
- Texturing, Caching and 2D Quad
- Comparisons
- Sequential Writes
Comparison with GPU-Based Algorithms

3-6x faster than prior GPU-based algorithms!
Sorting on Recent GPGPU APIs

- Use shared memory to perform local sorting
  - Bitonic sorting can be implemented in $O(n \log n)$ memory accesses
  - Memory access patterns similar to FFT butterfly networks
Preliminary performance of our algorithms:
• 4-8x faster than CUFFT on GTX280
• 2x faster using G92 than CUFFT on GTX280
• 10-30x faster than Intel MKL on high-end quad-core CPUs
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 $R \approx \sqrt{TN}$
Salzberg Analysis

If $N=100\text{GB}$, $T=2\text{MB}$, then $R \approx 230\text{MB}$

Large data sorting is inefficient on CPUs

$R \gg \text{CPU cache sizes} – \text{memory latency}$
External memory sorting

External memory sorting on CPUs can have low performance due to:
- High memory latency
- Or low I/O performance

Our algorithm:
- Sorts large data arrays on GPUs
- Perform I/O operations in parallel on CPUs
GPUTeraSort

Reader ➔ RAM ➔ Disks

Key-Pointer Gen ➔ CPU ➔ RAM

Sorter ➔ RAM ➔ GPU ➔ Video RAM

Reorder ➔ CPU ➔ RAM

Writer ➔ Disks ➔ DMA
I/O Performance

Salzberg Analysis:
100 MB Run Size

![Graph showing I/O performance for different RAID configurations](image-url)
I/O Performance

Salzberg Analysis:
100 MB Run Size

Pentium IV:
25MB Run Size

Less work and only 75% I/O 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 the Terabyte sorter in 2006

World’s best performance/$ system in 2006
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 in 2006
Conclusions

- Benchmarking is important for both IHVs and ISVs
  - Sorting is an important workload

- Design better algorithms on GPUs
  - Do not try to remap serial algorithms
  - Design scalable primitives (eg. scans), libraries (eg. MapReduce) and exploit them for adaptive, general and external memory sorting algorithms
Conclusions

- Exploit the memory models
  - FFT algorithms currently achieving over 0.25 TFLOPS per GPU
  - Applicable to many scientific computing algorithms on many-core architectures

- Novel external memory sorting algorithm as a scalable solution
  - Achieves high I/O performance on CPUs
  - Best performance/price solution – world’s fastest sorting system in 2006
GPU Roadmap

- GPUs are becoming more general purpose
  - Fewer limitations in Microsoft DirectX11 API
    - IEEE floating point support and optional double support
    - Integer instruction support,
    - More programmable stages, etc.
  - Significant advance in performance

- GPUs are being widely adopted in commercial applications
  - Image and media processing, signal processing, finance, etc.
Call to Action

- Pay attention to data parallelism
- Don’t put all your eggs in the multi-core basket
- If you want TeraOps – go where they are
- If you want memory bandwidth – go where the memory bandwidth is.
- CPU-GPU gap is widening
Acknowledgements

Collaborators:

- Jim Gray (Microsoft Research)
- Ming Lin (UNC)
- Qiong Luo (HKUST)
- Dinesh Manocha (UNC)
- Peter-Pike Sloan (Disney Interactive)

- Brandon Lloyd (Microsoft)
- Yuri Dotsenko (Microsoft)
- Chas Boyd (Microsoft)
- Burton Smith (Microsoft)
- John Manferdelli (Microsoft)
Acknowledgements

Supporters:
- Fred Brooks (UNC Chapel Hill)
- Craig Mundie (Microsoft)
Thank You

Questions or Comments?

nagag@microsoft.com