Analysis-driven Engineering of Comparison-based Sorting Algorithms on GPUs

32nd ACM International Conference on Supercomputing · June 17, 2018

Ben Karsin¹ · karsin@hawaii.edu
Volker Weichert² · weichert@cs.uni-frankfurt.de
Henri Casanova¹ · henric@hawaii.edu
John Iacono³ · john.iacono@ulb.ac.be
Nodari Sitchinava¹ · nodari@hawaii.edu

¹Department of ICS, University of Hawaii at Manoa
²Goethe University Frankfurt
³Département d’Informatique, Université Libre de Bruxelles

Work supported by the National Science Foundation under grants 1533823 and 1745331
Sorting: A fundamental problem

- Sorting is a building block
  - Used by countless algorithms...
Sorting: A fundamental problem

- Sorting is a building block
  - Used by countless algorithms...

\[ O(N) \]

\[ O(\log N) \]
Sorting: A fundamental problem

- Sorting is a building block
  - Used by countless algorithms...
Sorting: A fundamental problem

- Sorting is a building block
  - Used by countless algorithms...
Sorting: A fundamental problem

- Sorting is a building block
  - Used by countless algorithms...

- Many solutions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Stable?</th>
<th>Inplace?</th>
<th>Growth rate to sort N items</th>
<th>Extra space</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>no</td>
<td>yes</td>
<td>$N^2$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>insertion sort</td>
<td>yes</td>
<td>yes</td>
<td>between $N$ and $N^2$</td>
<td>1</td>
<td>depends on order of input keys</td>
</tr>
<tr>
<td>shellsort</td>
<td>no</td>
<td>yes</td>
<td>$N^{6/5}$?</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>quicksort</td>
<td>no</td>
<td>yes</td>
<td>$N \lg N$</td>
<td>$\lg N$</td>
<td>probabilistic guarantees, depend on distribution of input key values</td>
</tr>
<tr>
<td>3-way quicksort</td>
<td>no</td>
<td>yes</td>
<td>between $N$ and $N \lg N$</td>
<td>$\lg N$</td>
<td></td>
</tr>
<tr>
<td>mergesort</td>
<td>yes</td>
<td>no</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td>heapsort</td>
<td>no</td>
<td>yes</td>
<td>$N \lg N$</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Graphics Processing Units

- Designed for **high throughput**
- Extremely Parallel
  - Thousands of cores
- Huge performance potential
  - Lots of **application** research
  - No standard performance model
NVIDIA GPU

- Streaming Multiprocessors (SMs)
  - < 20 per GPU
  - < 200 cores each
NVIDIA GPU

- Streaming Multiprocessors (SMs)
  - < 20 per GPU
  - < 200 cores each

- Memory Hierarchy
  - User-controlled
  - Different scope
NVIDIA GPU

- Streaming Multiprocessors (SMs)
  - < 20 per GPU
  - < 200 cores each

- Memory Hierarchy
  - User-controlled
  - Different scope

- Thread organization
  - Cores share logic

- Need lots of parallelism!
Thread Organization

[Diagram of GPU architecture showing SMs and global memory]

Ben Karsin – A Performance Model for GPU Architectures
Thread Organization

- Threads are grouped into *thread-blocks*
  - $b$ threads
  - Run on the SM
- Threads are grouped into *thread-blocks*
  - $b$ threads
  - Run on the SM
- Groups of $w = 32$ form a *warp*
  - execute in ‘SIMT’ lockstep
Memory Hierarchy

- 3 levels with different:
  - Access scope
  - Capacity
  - Access pattern
  - Latency
  - Peak bandwidth
Global Memory

- Large (up to 32 GB)
- Shared by all threads
- Slow
- “Blocked” accesses
  - I/O model
Global Memory Access Pattern

- **Warp** - 32 threads execute in lockstep
  - Access global memory together

- Warp is a single unit
- 1 operation accesses 32 elements
- Just like disk accesses in ’I/O’ model ($B = 32$)
Shared Memory

- Small (48-64 KB per SM)
- Private to SM
  - User defines sharing
- 5 – 10 × faster
- Unique access pattern
  - organized into banks
Shared Memory Access Pattern

- Stored across $w$ memory banks

```
A
```

```
Bank 1
Bank 2
Bank 3
Bank 4
```
Shared Memory Access Pattern

- Separate banks accessed concurrently

Shared memory

<table>
<thead>
<tr>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
<th>Bank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
Shared Memory Access Pattern

- Threads accessing same bank = Bank conflict
- Serialize access

![Shared Memory Access Pattern Diagram](image)

<table>
<thead>
<tr>
<th>Bank 1</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Registers

- Small (255 per thread)
- Private to thread
- Fastest
- Random access
- Must be “static”
  - known at compile time
Talk Outline

- Motivation/background
- GPU overview
  - Memory hierarchy
- State-of-the-art GPU sorting
- Our multiway mergesort (GPU-MMS)
  - Optimizations
- Performance results
- Conclusions & future work
State-of-the-art GPU sorting

- Modern GPU (MGPU)
  - Pairwise mergesort

- CUB
  - Radix sort
  - Limited application

- Thrust
  - Changes algorithm based on input type
  - Comes with CUDA compiler

- All highly engineered and optimized for hardware
- Change parameters based on hardware detected
MGPU mergesort

- Pairwise mergesort
- $E$ elements per thread
MGPU mergesort

- Pairwise mergesort
- \( E \) elements per thread
- \( b \) threads per thread-block
MGPU mergesort

- Pairwise mergesort
- $E$ elements per thread
- $b$ threads per thread-block
- Lots of parallelism
  - $\frac{N}{E}$ threads!
MGPU mergesort

- Each thread-block sorts $bE$ elements
MGPU mergesort

- Each thread-block sorts $bE$ elements
- Merge pairs of lists
MGPU mergesort

- Each thread-block sorts $bE$ elements
- Merge pairs of lists
- $\left\lceil \log \frac{N}{bE} \right\rceil$ merge rounds
  - $b$ and $E$ iare small constants
MGPU bottlenecks

- Global memory is the main bottleneck
- Unavoidable: $O(\log_2 N)$ merge rounds
Multiway mergesort

- Reduce global memory bottleneck
  - Merge $K$ lists at a time!

\[ \lceil \log_2 \frac{N}{B} \rceil \text{ merge rounds} \]

- Merging done in internal memory
  - Use a priority queue

\[ \log_2 \frac{N}{B} \text{ merge rounds} \]
Merging $K$ lists

- Use a heap
- Load blocks from each list
- Build min-heap on smallest items

```
1
  /\  \
 3 6
 / \ / \ \
7 4 8 7
  / \ / \ \
8 9 6 11
 / \  / \
9 11 12 19
 /   /   \
16 8 13 16
```

$K$
Merging $K$ lists

- Use a heap
- Buffer smallest item
- Heapify to find next smallest

Diagram:

```
        3
       / \  
      4   6
     / \  / \
   7   5 8
  / \ / \ / \ 
8 9 6 8 10 14
11 7 7 11 12 19
16   13 22 18
   K
   1
```
Merging $K$ lists

- Use a heap
- Output buffer when full
- Read block when needed
Parallel ’Block Heap’

- Warp shares a heap
- 32 threads all need work...

```
32
K
```
Parallel 'Block Heap'

- Each node has a sorted list
Parallel ’Block Heap’

- Each node has a sorted list
Parallel 'Block Heap'

- Each node has a sorted list
- Merge child nodes
  - All 32 threads work together
Parallel 'Block Heap'

- Each node has a sorted list
- Merge child nodes

```
  7 8 9 11
 /   \   /
/     \ / \
19 22 23 30 18 20 21 24 28 29 31 33 23 24 25 26
```

Smallest: 7
Largest: 33
Parallel ’Block Heap’

- Each node has a sorted list
- Merge child nodes
- Repeat on empty child

![Block Heap Diagram]

Ben Karsin – A Performance Model for GPU Architectures
Multiway mergesort (GPU-MMS) analysis

- Base case sorts $w^2$ elements
- Merge groups of $K$ lists per round
  - $\lceil \log_K \frac{N}{w^2} \rceil$ rounds
- No bank conflicts
- Perform merging of nodes in registers
Multiway mergesort (GPU-MMS) analysis

- Base case sorts $w^2$ elements
- Merge groups of $K$ lists per round
  - $\lceil \log_K \frac{N}{w^2} \rceil$ rounds
- No bank conflicts
- Perform merging of nodes in registers
- Not work-efficient
  - $\log w$ more register accesses
Multiway mergesort (GPU-MMS) analysis

- Base case sorts $w^2$ elements
- Merge groups of $K$ lists per round
  - $\lceil \log_K \frac{N}{w^2} \rceil$ rounds
- No bank conflicts
- Perform merging of nodes in registers
- Not work-efficient
  - $\log w$ more register accesses
- Low parallelism
  - Lots of shared memory used
  - Dependent operations
Pipelining merge steps

- Pre-search path to leaf
- Identify all nodes to be merged
Pipelining merge steps

- Pre-search path to leaf
- Identify all nodes to be merged

Output

Merge

1 2 4 5

7 9 11 17

19 22 23 30

18 20 21 24

28 29 31 33

23 24 25 26
Tuning $K$

- Small $K$: too many global memory access
- Large $K$: not enough parallelism
Sorting Performance

- Sorting integers on Maxwell GPU
Impact of Bank Conflicts

- Generate input that causes bank conflicts
  - GPU-MMS is unaffected
Different datatypes

- Increasing comparison work degrades performance
Conclusions

- Analysis helps us develop better GPU algorithms
- I/O-efficient techniques work well
  - Minimize global memory accesses
  - Don’t forget parallelism
Conclusions

- Analysis helps us develop better GPU algorithms
- I/O-efficient techniques work well
  - Minimize global memory accesses
  - Don’t forget parallelism

Future work

- Optimize GPU-MMS
  - Work efficient (open problem)
- Apply analysis methods to other algorithms
- How will future architectures change things?
Conclusions

- Analysis helps us develop better GPU algorithms
- I/O-efficient techniques work well
  - Minimize global memory accesses
  - Don’t forget parallelism
- Future work
  - Optimize GPU-MMS
    - Work efficient (open problem)
  - Apply analysis methods to other algorithms
  - How will future architectures change things?

Thank You!

GPU-MMS available: https://github.com/algoparc/GPU-MMS
Backup Slides
MGPU Merge phase

- Merge pairs of lists
  - Repeat until sorted
MGPU Merge phase

- Merge pairs of lists
  - Repeat until sorted
- Find thread-block partition
MGPU Merge phase

- Merge pairs of lists
  - Repeat until sorted
- Find thread-block partition
- Each thread-block loads partition into shared memory
MGPU Merge phase

- Merge pairs of lists
  - Repeat until sorted
- Find thread-block partition
- Each thread-block loads partition into shared memory
- And merges...

TB1

TB2

TB3

TB4

...
GPU-MMS Bottlenecks

- Mostly compute-bound

![Graph showing GPU-MMS performance bottlenecks]

- GMEM
- SMEM
- Sync
- Basecase
- Registers
Searching in global memory
Model results: MGPU mergesort

- Model is quite accurate!
- Shows that $E = 31$ is ideal for this GPU!
  - $(E = 15$ is hard-coded)
Hiding Latency

- $t_x$: average time per $x$ operation
  - Min $t_x \rightarrow$ max throughput
- But operations have latency...
Hiding Latency

- $t_x$: average time per $x$ operation
  - $\min t_x \rightarrow \max$ throughput
- But operations have latency...
- Multiplicity: $\mathcal{X}$
  - multiple threads per core

![Diagram showing core, threads, and memory with arrows indicating data flow.](image-url)
Hiding Latency

- $t_x$: average time per $x$ operation
  - min $t_x \rightarrow$ max throughput
- But operations have latency...
- Multiplicity: $\mathcal{X}$
  - multiple threads per core

thread sends request to slow memory

![Diagram showing a core with multiple threads requesting memory](image_url)
Hiding Latency

- $t_x$: average time per $x$ operation
  - min $t_x \rightarrow$ max throughput
- But operations have latency...
- Multiplicity: $\mathcal{X}$
  - multiple threads per core

switch out thread while it waits

```
threads

core

request

Memory
```
Hiding Latency

- $t_x$: average time per $x$ operation
  - min $t_x \rightarrow$ max throughput
- But operations have latency...
- Multiplicity: $\lambda$
  - multiple threads per core

schedule new thread to use core

threads

core

request

Memory
Hiding Latency

- $t_x$: average time per $x$ operation
  - $\min t_x \rightarrow \max$ throughput

- But operations have latency...

- Multiplicity: $\lambda$
  - multiple threads per core

issue more requests to saturate bandwidth

Ben Karsin – A Performance Model for GPU Architectures
Hiding Latency

- $t_x$: average time per $x$ operation
  - min $t_x \rightarrow$ max throughput
- But operations have latency...
- Instruction-level parallelism (ILP): $\mathcal{I}$
  - consecutive independent instructions
Hiding Latency

- $t_x$: average time per $x$ operation
  - min $t_x \rightarrow$ max throughput
- But operations have latency...
- Instruction-level parallelism (ILP): $\mathcal{I}$
  - consecutive independent instructions

thread requests memory element $X$
Hiding Latency

- \( t_x \): average time per \( x \) operation
  - \( \min t_x \rightarrow \text{max throughput} \)
- But operations have latency...
- Instruction-level parallelism (ILP): \( \mathcal{I} \)
  - consecutive independent instructions

next instruction requests \( Y \)

[Diagram showing a core requests \( X \) from memory]

Ben Karsin – A Performance Model for GPU Architectures
Hiding Latency

- $t_x$: average time per $x$ operation
  - $\min t_x \rightarrow \text{max throughput}$
- But operations have latency...
- Instruction-level parallelism (ILP): $I$
  - consecutive independent instructions

issue next request without waiting for $X$

![Diagram showing core, request Y, request X, and Memory]
Hiding Latency

- $t_x$: average time per $x$ operation
  - $\min t_x \rightarrow \max$ throughput
- But operations have latency...
- Instruction-level parallelism (ILP): $\mathcal{I}$
  - consecutive independent instructions

issue more requests to saturate bandwidth
Impact of $\mathcal{X}$ and $\mathcal{I}$ (global memory)

- Copy $2^{16}$ elts. in global memory *per thread*
- When $\mathcal{X} \cdot \mathcal{I} \geq 8$ is limited by bandwidth
Impact of $\mathcal{X}$ and $\mathcal{I}$ (global memory)

- Copy $2^{16}$ elts. in global memory *per thread*
- When $\mathcal{X} \cdot \mathcal{I} \geq 8$ is limited by bandwidth

![Graph showing the relationship between $\mathcal{X}$ (threads per core) and Average Runtime (ms) for different values of $\mathcal{I}$.

- $\mathcal{I} = 1$ (solid blue line)
- $\mathcal{I} = 2$ (dotted red line)
- $\mathcal{I} = 4$ (dashed yellow line)
- $\mathcal{I} = 8$ (solid green line)
- $\mathcal{I} = 16$ (dotted purple line)

The graph illustrates the average runtime in milliseconds for different values of $\mathcal{X}$ and $\mathcal{I}$, with a notable point at $\mathcal{X} \cdot \mathcal{I} = 8 \cdot 1$. 

Ben Karsin – A Performance Model for GPU Architectures
Impact of $\lambda$ and $I$ (global memory)

- Copy $2^{16}$ elts. in global memory *per thread*
- When $\lambda \cdot I \geq 8$ is limited by bandwidth

![Graph showing the impact of $\lambda$ and $I$](image)

- $\lambda \cdot I = 8 \cdot 1$
- $\lambda \cdot I = 4 \cdot 2$
Impact of $\mathcal{X}$ and $\mathcal{I}$ (global memory)

- Copy $2^{16}$ elts. in global memory *per thread*
- When $\mathcal{X} \cdot \mathcal{I} \geq 8$ is limited by bandwidth

![Graph showing the impact of $\mathcal{X}$ and $\mathcal{I}$ on average runtime.](image)
Time per memory access

- Increasing \((X \cdot I)\):
  - Reduce latency
  - Until peak bandwidth reached

- Parameters for each type of memory:
  - \(L_x\) - memory access latency (clock cycles)
  - \(B_x\) - peak bandwidth
    - peak operations per clock cycle, per core

- Reduce latency until bandwidth reached:

\[
t_x = \max \left( \frac{1}{B_x}, \left\lceil \frac{L_x}{X \cdot I} \right\rceil \right)
\]
GPU Hardware Parameters

- Run benchmarks on 3 architectures
  - **ALGOPARC**: server in our lab
  - **GIBSON**: desktop with GPU
  - **UHHPC**: GPU node of UH cluster

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ALGOPARC</th>
<th>GIBSON</th>
<th>UHHPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GPU</td>
<td>Quadro M4000</td>
<td>GTX 770</td>
<td>K40</td>
</tr>
<tr>
<td>$P$ (total cores)</td>
<td>1664</td>
<td>1536</td>
<td>2880</td>
</tr>
<tr>
<td>$L_g$</td>
<td>269.5</td>
<td>267.6</td>
<td>291.2</td>
</tr>
<tr>
<td>$B_g$</td>
<td>0.0301</td>
<td>0.0279</td>
<td>0.0275</td>
</tr>
<tr>
<td>$L_s$</td>
<td>85.84</td>
<td>123.1</td>
<td>111.9</td>
</tr>
<tr>
<td>$B_s$</td>
<td>0.233</td>
<td>0.13</td>
<td>0.131</td>
</tr>
<tr>
<td>$L_r$</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>$B_r$</td>
<td>$\sim 1$</td>
<td>$\sim 1$</td>
<td>$\sim 1$</td>
</tr>
</tbody>
</table>