EFFICIENT STREAM COMPACCTION
ON WIDE SIMD MANY-CORE ARCHITECTURES

Markus Billeter
Ola Olsson
Ulf Assarsson

Chalmers University of Technology
Outline

• Why compact?
• Algorithm & Implementation
  – Contributions
  – Description
  – Analysis
• Results
• Bonus: Sorting & Prefix Sums
Compact?

• Remove invalid elements from an buffer
  – Predicate determines validity
  – Output: buffer containing only valid elements

• Building block for parallel algorithms
  – Pack sparse output
  – Parallel tree traversal and building
  – Parallel sorting
Previous Work

• Based on parallel prefix sums
  – Blelloch et al. [1990], Chatterjee et al. [1990]

  ![Diagram](image)

  – Recent GPU implementation in CUDPP 1.1
    • Sengupta et al [2007,2008]
Contributions

• Our implementation is
  – 12× faster than Geometry Shaders
  – At least 2.5× faster any other implementation we know of

• Uses negligible amount of extra storage
  – Not in-place though
Our view of a GPU

• $P$ virtual processors
  – Larger than the actual number of processors
  – Used to facilitate latency hiding
• SIMD width of $S$
• Wide Memory Bus
  – Generally need to access $S$ consecutive elements
  – No caches
• See paper for detailed explanation.
Actual Hardware

• Developed using an NVIDIA GTX280 GPU
  – With the CUDA 2.1 API, updated for CUDA 2.2!

• We used the following settings:
  – $P = 480$ virtual processors
    • 30 multiprocessors
    • 4 warps per multiprocessor
    • times 4 for latency hiding (empiric)
  – $S = 32$ (one warp)
    • Could use $S = 16, 64, ...$
Algorithm – General Idea

• Sequential algorithm – very simple
• Our approach
  – Number of independent processors
  – Large input set
  ⇒ Apply sequential algorithm to many independent pieces, and combine the results later
Algorithm: overview

I. Setup

• Have $N$ input elements
  – $N \gg P$
  – Number of output elements unknown

• Each processor:
  – Assigned chunk of data
  – Roughly equal size $\approx N/P$

• $O(1)$
Algorithm: overview

II. Count Elements

- Each processor:
  - count valid elements
- Independent chunks
  - Each processor has its own data
  ⇒ No synchronization
- $O\left(\frac{N}{PS} + \log S\right)$
Algorithm: overview

III. Find Offsets

- Find output offsets
  - Prefix sum over the element counts
- Constant number, $P$, of elements
  - E.g. 480 in our implementations
- $O(\log P)$
Algorithm: overview

IV. Copy Elements

• Each processor:
  – Copy valid elements in its chunk

• Source and destination known
  ⇒ no synchronization

• $O(N/(PS) \times \log S)$
SIMD compaction step

• Each iteration during compaction
  – Load $S = 32$ consecutive elements
  – Discard invalid elements
    • E.g. prefix sum (or POPC) over $S$ elements

  \[
  \begin{array}{cccccccccccc}
  A & x & x & B & C & x & x & x & x & D & x & E & F & G & H & x \\
  \end{array}
  \]

  – Store remaining elements
    • To CUDA shared memory
    • To CUDA global memory (output array)
Analysis

• Complexity (number of steps):
  – $O\left(\frac{N}{PS} \times \log S + \log P\right) \sim O(N)$

• SIMD Width tradeoff:
  – Large $S$ – increases compute ($\log S$)
    • i.e. if we pick $S = 1$, $P' = PS$ complexity is reduced to $O(N/P' + \log P')$
  – Small $S$ – worse memory access pattern
    • In CUDA: no coalesced memory reads

• $S = 32$ turns out to be good (empiric)
Population Count - POPC

• Count set bits in a word
  – Alternative to prefix sum in compaction step
  – If implemented in hardware, we get rid of $\log(S)$ factor

• Also need to broadcast result of predicate to all lanes/threads
  – Currently expensive in CUDA
  – Also, POPC is not (yet?) a hardware instruction
Optimization : 64bit fetches

• Optimization : fetch 32bit data in 64bit units
  – Increases bandwidth
  – Hardware specific
  – Limited to 32bit data

• Each iteration now handles $2 \times 32$ elements
  – Pass-through bandwidths are:

<table>
<thead>
<tr>
<th>32x</th>
<th>32 bit fetches</th>
<th>64 bit fetches</th>
<th>128 bit fetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (GB/s)</td>
<td>77.8</td>
<td>102.5</td>
<td>73.4</td>
</tr>
</tbody>
</table>
Variations

- Several variants available
  - Tradeoff: memory access vs. compute load
- Output
  - *Staged*: as described. Valid elements are placed in shared memory
  - *Scattered*: bypasses compaction to shared memory
  - *Buffered*: accumulates $S$ elements in shared memory
Variations (contd.)

• Staging and buffering is not free
  – Higher computational load
  – On an GTX280: buffering never viable
  – Sweeney: 1 byte / 1 op
    ⇒ 1 op / 1 byte. Address logic op = ordinary op

• Dynamically choose: staging or scattering
  – Know ratio of valid elements from count
  – Heuristic (≈manually tweaked) threshold
Variants – Comparison

Stream Compaction (4M elements – 32bit)

Stream Compaction (4M elements – 64bit)
Results

- **2.9× speedup vs. CUDPP**
  - 2.5× without computing extra flags
- **Less auxiliary memory**
  - Order of $P \approx 500$ elements
- **We can compact 64bit elements faster than 32bit in CUDPP**
  - 2× data
Bonus: Prefix Sum

• Easier than compaction
  – Number of output elements is equal to inputs
    ⇒ perfect coalescing when reading and writing!
• Results: 220M elements
  – 32bit: 880Mbyte data
  – 25.3ms
Bonus: Radix Sort – Part A

• “Stream split”
  – Compaction that places invalid elements in second half of the output buffer

• Radix Sort
  – Apply stream split once for each bit in the key
  – Can optimize further
    • “Plain” vs. “Optimized”
Bonus: Radix Sort – Part B

Until yesterday (Sunday) this slide would claim that we have the fastest Radix Sort implementation.

However: “Scalable Split & Sort Primitives”

– Poster by Suryakart Patidav and P.J. Narayanan here at HPG

seem to sort a few % faster.
Conclusions

• Efficient stream compaction
  – Approx. 3× speedup vs. older implementations
  – Handles 32bit, 64bit and 128 bit elements currently
• Configurable for newer hardware
  – Different compute/memory tradeoffs
• Related algorithms
  – Stream Split
  – Radix Sort - fastest for >500k elements (~15%)
  – Prefix Sum - fastest (~ 30%)
Acknowledgements

• Swedish Foundation for Strategic Research

Implementation will be available at

• http://www.cse.chalmers.se/~billeter/pub/pp

Thank you for your attention.