State of the Art in GPU Data-Parallel Algorithm Primitives

Mark Harris
NVIDIA
Stream Programming Model

- Many independent threads of execution
  - All running the same program

- Threads operate in parallel on separate inputs
  - Produce an output per input

- Works well when outputs depend on small, bounded input
Stream Parallelism

- One-to-one Input-output dependence (e.g., scalar)
Stream Parallelism

- Local neighborhood input-output dependence (e.g., stencil)
Beyond streaming

- GPUs are obviously really good at local and 1:1 dependences
  - But many applications have more complex dependencies
  - ... and variable output

- Global, dynamic input-output dependences are common
  - Sorting, building data structures
- Use efficient algorithm primitives for common patterns

---

Algorithm Examples

- Sort, computational geometry, finance
  - Modest control flow
  - Use/irregular data structures
  - Irregular communication between elements
- CPU Territory
  - General purpose features vital for software efficiency
  - Latency sensitive applications

---

All data, figures and product plans are preliminary and are subject to change without notice. Copyright © Intel Corporation 2006
Parallel Patterns

- Many parallel threads need to generate a single result value
  - “Reduce”

- Many parallel threads need to partition data
  - “Split”

- Many parallel threads, variable output per thread
  - “Compact” / “Allocate”
Reduce

- Global data dependence?

Input: 3.4 4.1 2.0 1.5 9.6 0.3 7.1 1.4

Output: 29.4
Parallel Reduction: Easy

- Repeated local neighborhood access: $O(\log n)$ reps
  - Static data dependences, uniform output
Parallel Patterns

- Many parallel threads need to generate a single result value
  - “Reduce”

- Many parallel threads need to partition data
  - “Split”

- Many parallel threads, variable output per thread
  - “Compact” / “Allocate”
Split

- Example: radix sort, building trees
Parallel Patterns

- Many parallel threads need to generate a single result value
  - “Reduce”

- Many parallel threads need to partition data
  - “Split”

- Many parallel threads, variable output per thread
  - “Compact” / “Allocate”
Compact

- Remove unneeded or invalid elements (blue)

Example: collision detection
Variable Output Per Thread: General Case

- Allocate Variable Storage Per Thread

Example: marching cubes
“Where do I write my output?”

- Split, compact and allocate require all threads to answer

- The answer is:
  “That depends on how much the other threads output!”

- “Scan” is an efficient, parallel way to answer this question
Parallel Prefix Sums (Scan)

- Given array \( A = [a_0, a_1, \ldots, a_{n-1}] \)
  and a binary associative operator \( \oplus \) with identity \( I \),

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

- Example: if \( \oplus \) is +, then

\[
\text{Scan}([3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]) = [0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22] \text{ (exclusive)}
\]

\[
\text{Scan}([3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]) = [3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25] \text{ (inclusive)}
\]
Segmented Scan

Segment Head Flags $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$

Input Data Array $\begin{bmatrix} 3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 \end{bmatrix}$

Segmented scan $\begin{bmatrix} 0 & 3 & 0 & 7 & 7 & 0 & 1 & 7 \end{bmatrix}$

- Segmented scan provides *nested parallelism*
  - Arrays can be dynamically subdivided and processed in parallel
- Enables algorithms such as parallel quicksort, sparse matrix-vector multiply, etc.

S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. “Scan Primitives for GPU Computing”. *Graphics Hardware 2007*
How fast?

- Bandwidth bound primitives
  - 1 add per element read/write
  - Scan and reduce at memory saturated rates

- Geforce GTX 280
  - Scan 11.8B elements/second (32-bit elements)
  - Bandwidth: Scan 138 GB/s; Reduce 152 GB/s

D. Merrill & A. Grimshaw “Parallel Scan for Stream Architectures”. Tech. Report CS2009-14, Department of Computer Science, University of Virginia.
Applications of Scan

- A simple and useful building block for many parallel apps:
  - Compaction
  - Radix sort
  - Quicksort (segmented scan)
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Run-length encoding
  - Allocation
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Summed area tables
  - And many more!

- (Interestingly, scan is unnecessary in sequential computing)
Compact using Scan

- Flag unneeded elements with zero:

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- Threads with flag == 1 use scan result as address for output:

  | Output | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Recent efficient approach:
Radix Sort / Split using Scan

\[
\begin{align*}
  i &= \text{index} \\
  b &= \text{current bit} \\
  d &= \text{invert } b \\
  f &= \text{scan}(d) \\
  t &= \text{numZeros} + i - f \\
  \text{out} &= b \? t : f
\end{align*}
\]

Current Digit: 0

\[
\begin{align*}
  i &= \text{index} \\
  0 &\quad 1 &\quad 2 &\quad 3 &\quad 4 &\quad 5 &\quad 6 &\quad 7 \\
  b &= \text{current bit} \\
  0 &\quad 0 &\quad 1 &\quad 1 &\quad 1 &\quad 0 &\quad 1 &\quad 1 &\quad 0 &\quad 0 &\quad 1 &\quad 1 &\quad 0 \\
  d &= \text{invert } b \\
  1 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 1 \\
  f &= \text{scan}(d) \\
  0 &\quad 1 &\quad 1 &\quad 1 &\quad 1 &\quad 1 &\quad 1 \\
  t &= \text{numZeros} + i - f \\
  3 &\quad 3 &\quad 4 &\quad 5 &\quad 6 &\quad 7 &\quad 8 &\quad 8 \\
  \text{out} &= b \? t : f \\
  0 &\quad 3 &\quad 4 &\quad 5 &\quad 6 &\quad 7 &\quad 1 &\quad 2 \\
  \text{Current Digit: 0} \\
  \text{numZeros}=3
\end{align*}
\]
## Radix Sort / Split using Scan

<table>
<thead>
<tr>
<th>i = index</th>
<th>0</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>b = current bit</td>
<td>010</td>
<td>000</td>
<td>110</td>
<td>001</td>
<td>111</td>
<td>011</td>
<td>101</td>
<td>011</td>
</tr>
</tbody>
</table>

**Current Digit:** 1

<table>
<thead>
<tr>
<th>d = invert b</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = scan(d)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- numZeros = 3

<table>
<thead>
<tr>
<th>t = numZeros + i - f</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = b ? t : f</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

```
000 001 101 010 110 111 011 011
```
Radix Sort / Split using Scan

\[ f = \text{scan}(d) \]
\[ b = \text{current bit} \]
\[ d = \text{invert } b \]
\[ f = \text{scan}(d) \]
\[ t = \text{numZeros} + i - f \]
\[ d = b \ ? t : f \]

<table>
<thead>
<tr>
<th>i = index</th>
<th>0</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>b = current bit</td>
<td>000</td>
<td>001</td>
<td>101</td>
<td>010</td>
<td>110</td>
<td>111</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>d = invert b</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f = scan(d)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>t = numZeros + i - f</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>d = b ? t : f</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Current Digit: 2
numZeros=5
CUDA Radix Sort

- Sort blocks in shared mem
  - 4-bit radix digits, so 4 split operations per digit
- Compute offsets for each block using prefix sum
- Scatter results to offset location


Fig. 7. Sorting rates for several GPU-based methods on an 8800 Ultra.
Faster Radix Sort Via Rigorous Analysis

- Meta-strategy leads to ultra-efficient bandwidth use and computation
  - Optimized data access saturates bandwidth
  - Combine multiple related compact ops into a single scan
  - Reduce-then-scan strategy

- Radix sort 1B keys/s on Fermi! (Up to 3.8x vs. Satish et al.)

- See Duane Merrill’s GTC talk

“Optimization for Ninjas: A Case Study in High-Performance Sorting”


Open Source: http://code.google.com/p/back40computing
Designing Sorting Algorithms for GPUs

- Algorithms should expose regular fine-grained parallelism
  - scan used to regularize
  - In merging, use divide-and-conquer to increase parallelism close to tree root (Satish et al. 2007)
  - Optimize memory access granularity first - max bandwidth is key

- Comparison vs. Key-manipulation
  - Comparison sort = \(O(n \log n)\), works for any criterion
  - Radix sort = \(O(n)\), but requires numeric key manipulation

- Important to handle key-value pairs
  - Pointer-as-value enables sorting big objects
Comparison Sorting Algorithms

- Use comparison sorting when key manipulation not possible
  - Variations on parallel divide-and-conquer approach
- Sample Sort (Fastest current comparison-based sort)
  - Leischner, Osipov, Sanders. IPDPS 2010
- Merge Sort
  - Satish, Harris, Garland. IPDPS 2009
- Parallel Quicksort
  - Cederman & Tsigas, Chalmers U. of Tech. TR#2008-01
  - Sengupta, Harris, Zhang, Owens, Graphics Hardware 2007
Building Trees

- The split primitive can be applied to any Boolean criterion...

- Hierarchies built by splitting on successive spatial partitions
  - E.g. splitting planes

- Trees: special case of sorting!
Bounding Volume Hierarchies

- Bounding Volume Hierarchies:
  - Breadth-first search order construction
  - Use space-filling “Morton curve” to reduce BVH construction to sorting
  - Requires 2 $O(n)$ radix sorts

Eurographics 2009

“LBVH” - Linear Bounding Volume Hierarchies
HLBVH

- Improvement over LBVH
  - 2-3x lower computation, 10-20x lower bandwidth
  - 2-4x more compact tree memory layout


<table>
<thead>
<tr>
<th>Scene</th>
<th># of Triangles</th>
<th>LBVH</th>
<th>HLBVH</th>
<th>HLBVH + SAH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>max</td>
<td>min</td>
</tr>
<tr>
<td>Armadillo</td>
<td>345k</td>
<td>61 ms</td>
<td>27 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>Stanford Dragon</td>
<td>871k</td>
<td>98 ms</td>
<td>36 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>Happy Buddha</td>
<td>1.08M</td>
<td>117 ms</td>
<td>43 ms</td>
<td>32 ms</td>
</tr>
<tr>
<td>Turbine Blade</td>
<td>1.76M</td>
<td>167 ms</td>
<td>54 ms</td>
<td>42 ms</td>
</tr>
<tr>
<td>Hair Ball</td>
<td>2.88M</td>
<td>241 ms</td>
<td>95 ms</td>
<td>83 ms</td>
</tr>
</tbody>
</table>
**k-d Trees**

- Spatial partition for organizing points in $k$-dimensional space
  - Commonly used in ray tracing, photon mapping, particle simulation

- Breadth-first search order
  - Parallelizes on nodes at lower tree levels (many nodes)
  - Parallelizes on geometric primitives at upper tree levels (few nodes)

Breadth-First Search Order

- BFS order construction maximizes parallelism

- Breadth First:

- Depth First:
Memory-Scalable Hierarchies

- Breadth-first search order has high storage cost
  - Must maintain and process lots of data simultaneously

- Solution: partial breadth-first search order
  - Limit number of parallel splits
  - Allows scalable, out-of-core construction
  - Works for kD-trees and BVH

Parallel Hashing

- Dense data structure for storing sparse items
  - With fast construction and fast random access
- Hybrid multi-level, multiple-choice ("cuckoo") hash algorithm
  - Divide into blocks, cuckoo hash within each block in shared memory

Cuckoo Hashing

- **Sequential insertion:**
  1. Try empty slots first
  2. Evict if none available
  3. Evicted key checks its other locations
  4. Recursively evict

- **Assume impossible after** \( O(lg n) \) **iterations**
  - Rebuild using new hash functions

Pagh and Rodler [2001]
Cuckoo Hashing

- Sequential insertion:
  1. Try empty slots first
  2. Evict if none available
  3. Evicted key checks its other locations
  4. Recursively evict
- Assume impossible after $O(\lg n)$ iterations
  - Rebuild using new hash functions

Pagh and Rodler [2001]
Cuckoo Hashing

- Sequential insertion:
  1. Try empty slots first
  2. Evict if none available
  3. Evicted key checks its other locations
  4. Recursively evict

- Assume impossible after $O(\lg n)$ iterations
  - Rebuild using new hash functions

Pagh and Rodler [2001]
GPU Parallel Hash Performance

List Ranking

- Traverse a linked list and assign rank to each node
  - Rank = order in list
- Difficult for random lists due to irregular memory access
  - Pointer chasing
- Recursive list subdivision
  - Helman-JáJá algorithm

Implementing Scan: Strategies
Scan is recursive

- Build large scans from small ones
  - In CUDA, build array scans from thread block scans
  - Build thread block scans from warp scans

- One approach: “Scan-Scan-Add”
1. Scan All Sub-blocks

2. Scan Block Sums

3. Add Block Offsets
Improvement: Reduce-then-Scan

- Scan-Scan-Add requires 2 reads/writes of every element
- Instead, compute block sums first, use as prefix for block scans

- 25% lower bandwidth

- Dotsenko et al. 2008,
  Billeter et al. 2009,
  Merrill & Grimshaw 2009
1. Reduce All Sub-Blocks

2. Scan Block Sums

3. Scan Blocks, using block sums as prefixes
Limit recursive steps

- Can use these techniques to build scans of arbitrary length
  - In CUDA, each recursive level is another kernel call

- Better: iterative scans of consecutive chunks
  - Each thread block scans many chunks
  - Prefix each chunk with sum from of all previous chunks

- Limiting to 2-level scan in this way is much more efficient
  - Merrill and Grimshaw, 2009
Implementing Scan In CUDA
Scan Implementation in CUDA

- In the following examples:
  - all pointers assumed to point to CUDA __shared__ memory
  - The following variable definitions are assumed (for brevity):

```c
int i = threadIdx.x;
int lane = I & (warpSize - 1);
int warp = i / warpSize;
int n = blockDim.x;
```
Sequential Inclusive Scan Code

```c
int scan(int *p, int n) {
    for (int i=1; i<n; ++i) {
        p[i] = p[i-1]+p[i];
    }
}
```

- Parallel scan needs to parallelize the loop
  - Relies on associativity of the operator
Simple Parallel Inclusive Scan Code

```c
__device__ int scan(int *p) {
    for (int offset=1; offset<n; offset*=2) {
        int t;
        if (i>=offset) t = p[i-offset];
        __syncthreads();
        if(i>=offset) p[i]= t + p[i];
        __syncthreads();
    }
}
```
Warp Speed

- **Warp** is physical unit of CUDA parallelism
  - 32 threads that execute instructions synchronously

- Synchronicity of warps can be leveraged for performance
  - When sharing data within a warp, don’t need `__syncthreads()`

- “Warp-Synchronous Programming”
  - (Powerful, but take care to avoid race conditions!)

Intra-warp Exclusive Scan Code

```c
__device__ int scan_warp(volatile int *p) {
    if (lane>= 1) p[i] = p[i-1] + p[i];
    if (lane>= 2) p[i] = p[i-2] + p[i];
    if (lane>= 4) p[i] = p[i-4] + p[i];
    if (lane>= 8) p[i] = p[i-8] + p[i];
    if (lane>=16) p[i] = p[i-16] + p[i];
    return (lane>0) ? p[i-1] : 0;
}
```
Intra-block Scan using Intra-warp Scan

```c
__device__ int block_scan(int* p) {
    int prefix = scan_warp(p);
    __syncthreads();
    if (lane == warpSize - 1) p[warp] = prefix + x;
    __syncthreads();
    if (warp == 0) p[i] = scan_warp(p);
    __syncthreads();
    return prefix + p[warp];
}
```
Binary Scan

- Often need to scan 1-bit values
  - Stream compact: scan true/false flags
  - Split / Radix Sort: scan 1-bit flags

- Fermi GPU architecture provides efficient 1-bit warp scan
  - int __ballot(int p): 32-bit “ballot” of t/f p from whole warp
  - int __popc(int x): count number of 1 bits in x

- Combine with a per-thread mask to get 1-bit warp scan
  - Or with no mask for 1-bit warp count
Binary Warp Scan Code

```c
__device__ unsigned int lanemask_lt () {
    int lane = threadIdx.x & (warpSize-1);
    return (1 << lane) - 1;
}

__device__ int warp_binary_scan(bool p) {
    unsigned int b = __ballot(p);
    return __popc(b & lanemask_lt());
}
```
Block Binary Scan Performance

- Substitute binary warp-scan in block_scan

Binary Reduction

- Count the number of true predicates for all threads in block
  - int __syncthreads_count(int p);
  - Also __syncthreads_and() and __syncthreads_or()

- Works like __syncthreads(), but counts non-zero $p$

- 2x faster than 32-bit reduction
Parallel Primitive Libraries
No need to re-implement

- Open source libraries under active development
  - CUDPP: CUDA Data-Parallel Primitives library
    - [http://code.google.com/p/cudpp](http://code.google.com/p/cudpp) (BSD License)
  - Thrust
    - [http://code.google.com/p/thrust](http://code.google.com/p/thrust) (Apache License)
CUDPP

- C library of high-performance parallel primitives for CUDA
  - M. Harris (NVIDIA), J. Owens (UCD), S. Sengupta (UCD), A. Davidson (UCD), S. Tzeng (UCD), Y. Zhang (UCD)

- Algorithms
  - cudppScan, cudppSegmentedScan, cudppReduce
  - cudppSort, cudppRand, cudppSparseMatrixVectorMultiply

- Additional algorithms in progress
  - Graphs, more sorting, trees, hashing, autotuning
CUDPP Example

CUDPPConfiguration config = { CUDPP_SCAN,
      CUDPP_ADD, CUDPP_FLOAT, CUDPP_OPTION_FORWARD };

CUDPPHandle plan;
CUDPPResult result = cudppPlan(&plan,
      config,
      numElements,
      1, 0);

cudppScan(plan, d_odata, d_idata, numElements);
Thrust

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)

- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`

- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
   - Etc.
Thrust Example

// generate 16M random numbers on the host
thrust::host_vector<int> h_vec(1 << 24);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// transfer data to the device
thrust::device_vector<int> d_vec = h_vec;

// sort data on the device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
Conclusion: so much to be done!
“In general, the problem of defining parallel-friendly data structures that can be efficiently created, updated, and accessed is a significant research challenge... The toolbox of efficient data structures and their associated algorithms on scalar architectures like the CPU remains significantly larger than on parallel architectures like the GPU.”

-- Alcantara et al. “Real-Time Parallel Hashing on the GPU”
See These Talks!

- **Duane Merrill:**
  - Optimization for Ninjas: A Case Study in High-Performance Sorting
  - Wednesday, 3pm (Room D)

- **Nathan Bell:**
  - High-Productivity CUDA Development with the Thrust Template Library
  - Thursday, 11am (Marriott Ballroom)

- **Jared Hoberock:**
  - Thrust by Example: Advanced Features and Techniques
  - Thursday, 2pm (Room B)
Thank You!

- Duane Merrill, David Luebke, John Owens, CUDPP-dev team, Nathan Bell, Jared Hoberock, Michael Garland

- Questions/Feedback: mharris@nvidia.com
Scan Literature (1)

Pre-GPU

- First proposed in APL by Iverson (1962)
- Used as a data parallel primitive in the Connection Machine (1990)
  - Feature of C* and CM-Lisp
- Guy Blelloch popularized scan as a primitive for various parallel algorithms
  - Blelloch, 1990, “Prefix Sums and Their Applications”

Post-GPU

- $O(n \log n)$ work GPU implementation by Daniel Horn (GPU Gems 2)
  - Applied to Summed Area Tables by Hensley et al. (EG05)
- $O(n)$ work GPU scan by Sengupta et al. (EDGE06) and Greß et al. (EG06)
- $O(n)$ work & space GPU implementation by Harris et al. (2007)
Scan Literature (2)

- Sengupta et al. segmented scan, radix sort, quicksort (Graphics Hardware 2007)
- Sengupta et al. warp scan (NV Tech report 2008)
  - Extended in *Scientific Computing with Multicore and Accelerators*, Ch. 19. 2011
- Dotsenko et al. reduce-then-scan (ICS 2008)
- Billeter et al. efficient compact (HPG 2009)
- Satish et al. radix sort (IPDPS 2009)
- Merrill & Grimshaw, efficient GPU scan (UVA Tech Rep. 2009)
- Merrill & Grimshaw, efficient radix sort (UVA Tech Rep. 2010)
- Harris & Garland, binary scan (GPU Computing Gems 2, 2011)