CS 380 - GPU and GPGPU Programming
Lecture 26: Parallel Scan Bank Conflicts; 
Shuffle Instructions; 
Warp Synchronous Programming

Markus Hadwiger, KAUST
Reading Assignment #14 (until Dec 2)

Read (required):

• CUDA Cooperative Groups (Volta + Turing)
  - https://devblogs.nvidia.com/cooperative-groups/

• Programming Tensor Cores in CUDA 10/9

• CUDA 10/9 Features Revealed: Volta, Turing, cooperative groups, and more

Read (optional):

• Warp-aggregated atomics

• Fast Histograms Using Shared Atomics on Maxwell
Semester Project Presentation Event

Thursday, Dec 12, 12:30 – 14:30, room tba
  • We’ll get lunch, cookies, and coffee!

Presentation slots < 10 min
$O(n \log n)$ Scan

- **Step efficient** (log $n$ steps)
- **Not work efficient** ($n \log n$ work)
- **Requires barriers at each step** (WAR dependencies)
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.
A First-Attempt Parallel Scan Algorithm

1. (previous slide)

2. Iterate \( \log(n) \) times: Threads \( \text{stride} \) to \( n \): Add pairs of elements \( \text{stride} \) elements apart. Double \( \text{stride} \) at each iteration. (note must double buffer shared mem arrays)

**Iteration #1**

Stride = 1

- Active threads: \( \text{stride} \) to \( n-1 \) (\( n-\text{stride} \) threads)
- Thread \( j \) adds elements \( j \) and \( j-\text{stride} \) from T0 and writes result into shared memory buffer T1 (ping-pong)
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate $\log(n)$ times: Threads stride to $n$: Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

---

Iteration #2
Stride = 2
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate $\log(n)$ times: Threads stride to $n$: Add pairs of elements stride elements apart. Double stride at each iteration. (note must double buffer shared mem arrays)

<table>
<thead>
<tr>
<th>Iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride = 4</td>
</tr>
</tbody>
</table>
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate \( \log(n) \) times: Threads \( \text{stride} \) to \( n \): Add pairs of elements \( \text{stride} \) elements apart. Double \( \text{stride} \) at each iteration. (note must double buffer shared mem arrays)

3. Write output to device memory.
Work Efficiency Considerations

- The first-attempt Scan executes $\log(n)$ parallel iterations
  - Total adds: $n \cdot (\log(n) - 1) + 1 \rightarrow O(n\log(n))$ work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does $n$ adds
  - A factor of $\log(n)$ hurts: 20x for $10^6$ elements!

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency
Typical Parallel Programming Pattern

- 2 log(n) steps
Typical Parallel Programming Pattern

- $2 \log(n)$ steps
$O(n)$ Scan [Blelloch]

- Work efficient ($O(n)$ work)
- Bank conflicts, and lots of 'em
Build the Sum Tree

T 3 1 7 0 4 1 6 3

Assume array is already in shared memory
Build the Sum Tree

Iterate log(n) times. Each thread adds value stride / 2 elements away to its own value.

Each ✖️ corresponds to a single thread.
Build the Sum Tree

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
</table>

Stride 2

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
</table>

Stride 4

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>14</th>
</tr>
</thead>
</table>

Iteration 2, \( n/4 \) threads

Each \( \odot \) corresponds to a single thread.

Iterate \( \log(n) \) times. Each thread adds value \( stride / 2 \) elements away to its own value.
Build the Sum Tree

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
</table>
**Stride 2**
<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
</table>
**Stride 4**
<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>14</th>
</tr>
</thead>
</table>
**Stride 8**
|   | T | 3 | 4 | 7 | 11| 4 | 5 | 6 | 25|

Iteration $\log(n)$, 1 thread

Each $\odot$ corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering.
Down-Sweep Variant 1: Exclusive Scan

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.
# Build Scan From Partial Sums

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 0 |
Build Scan From Partial Sums

Stride 8

Iteration 1
1 thread

Each $\oplus$ corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value and sets the value $\text{stride}$ elements away to its own previous value.
Build Scan From Partial Sums

Iterate log(n) times. Each thread adds value \( \text{stride} / 2 \) elements away to its own value and sets the value \( \text{stride} / 2 \) elements away to its own previous value.

Each \( \oplus \) corresponds to a single thread.
Build Scan From Partial Sums

Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(n).
Total work: 2 * (n-1) adds = O(n)  Work Efficient!
Down-Sweep Variant 2: Inlusive Scan

| T | 3 | 4 | 7 | 11 | 4 | 5 | 6 | 25 |

We now have an array of partial sums. Let’s propagate the sums back.
Build Scan From Partial Sums

Stride 8

no operation

Each $\oplus$ corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value. First element adds zero.
Build Scan From Partial Sums

Iterate $\log(n)$ times. Each thread adds value $\text{stride} / 2$ elements away to its own value. First element adds zero.

Each + corresponds to a single thread.

Paralle08 – Control Flow  Hendrik Lensch and Robert Strzodka
Build Scan From Partial Sums

Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 \times \log(n)$.
Total work: $< 2 \times (n-1)$ adds = $O(n)$ \textbf{Work Efficient!}
Application to Large Arrays

Initial Array of Arbitrary Values

Scan Block 0  Scan Block 1  Scan Block 2  Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum $i$ to All Values of Scanned Block $i+1$

Final Array of Scanned Values
Scan papers


- Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.


Bank Conflicts in Scan
- Non-power-of-two -
Initial Bank Conflicts on Load

- Each thread loads two shared mem data elements
- Tempting to interleave the loads
  \[
  \text{temp}[2 \times \text{thid}] = \text{g_idata}[2 \times \text{thid}]; \\
  \text{temp}[2 \times \text{thid}+1] = \text{g_idata}[2 \times \text{thid}+1];
  \]
- Threads: \((0,1,2,\ldots,8,9,10,\ldots)\) \(\rightarrow\) banks: \((0,2,4,\ldots,0,2,4,\ldots)\)
- Better to load one element from each half of the array
  \[
  \text{temp}[\text{thid}] = \text{g_idata}[\text{thid}]; \\
  \text{temp}[\text{thid} + (n/2)] = \text{g_idata}[\text{thid} + (n/2)];
  \]
Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(0,8) access bank 0

First iteration: 2 threads access each of 8 banks.

Each ◉ corresponds to a single thread.
Like-colored arrows represent simultaneous memory accesses
Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(1,9) access bank 2, etc.

First iteration: 2 threads access each of 8 banks.

Each ◀️ corresponds to a single thread.
Like-colored arrows represent simultaneous memory accesses.
Bank Conflicts in the tree algorithm

- 2\textsuperscript{nd} iteration: even worse!
  - 4-way bank conflicts; for example:
    Th(0, 4, 8, 12) access bank 1, Th(1, 5, 9, 13) access Bank 5, etc.

Each \(\odot\) corresponds to a single thread.
Like-colored arrows represent simultaneous memory accesses
Scan Bank Conflicts (1)

- A full binary tree with 64 leaf nodes:

<table>
<thead>
<tr>
<th>Scale (s)</th>
<th>Thread addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62</td>
</tr>
<tr>
<td>2</td>
<td>0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60</td>
</tr>
<tr>
<td>4</td>
<td>0 8 16 24 32 40 48 56</td>
</tr>
<tr>
<td>8</td>
<td>0 16 32 48</td>
</tr>
<tr>
<td>16</td>
<td>0 32</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conflicts</th>
<th>Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way</td>
<td>0 2 4 6 8 10 12 14 0 2 4 6 8 10 12 14 0 2 4 6 8 10 12 14</td>
</tr>
<tr>
<td>4-way</td>
<td>0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 12</td>
</tr>
<tr>
<td>4-way</td>
<td>0 8 0 8 0 8 0 8</td>
</tr>
<tr>
<td>4-way</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>2-way</td>
<td>0 0</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Multiple 2-and 4-way bank conflicts**
- **Shared memory cost for whole tree**
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write ($s[a] += s[b]$)
  - $6 \times (2+4+4+4+2+1) = 102$ cycles
  - 36 cycles if there were no bank conflicts ($6 \times 6$)
Scan Bank Conflicts (2)

- It’s much worse with bigger trees!
- A full binary tree with 128 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)

<table>
<thead>
<tr>
<th>Scale (s)</th>
<th>Thread addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 96 104 108 112 116 120 122</td>
</tr>
<tr>
<td>4</td>
<td>0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120</td>
</tr>
<tr>
<td>8</td>
<td>0 16 32 48 64 80 96 112</td>
</tr>
<tr>
<td>16</td>
<td>0 32 64 96</td>
</tr>
<tr>
<td>32</td>
<td>0 64</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conflicts</th>
<th>Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-way</td>
<td>0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 12 0 4 8 10</td>
</tr>
<tr>
<td>8-way</td>
<td>0 8 0 8 0 8 0 8 0 8 0 8 0 8 0 8 0 8</td>
</tr>
<tr>
<td>8-way</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>4-way</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>2-way</td>
<td>0 0</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
</tr>
</tbody>
</table>

- Cost for whole tree:
  - \(12 \times 2 + 6 \times (4+8+8+4+2+1) = 186\) cycles
  - 48 cycles if there were no bank conflicts! \(12 \times 1 + (6 \times 6)\)
Bank Conflicts in the tree algorithm

- We can use padding to prevent bank conflicts
  - Just add a word of padding every 16 words:
- No more conflicts! 32 for full warps!

Now, within a 16-thread half-warp, all threads access different banks.
32-thread full warp!
(Note that only arrows with the same color happen simultaneously.)
Use Padding to Reduce Conflicts

- This is a simple modification to the last exercise

- After you compute a shared mem address like this:
  \[ \text{Address} = \text{stride} \times \text{thid}; \]

- Add padding like this:
  \[ \text{Address} += (\text{Address} \gg 4); \quad // \text{divide by NUM_BANKS} \]

- This removes most bank conflicts
  - Not all, in the case of deep trees
Fixing Scan Bank Conflicts

- Insert padding every NUM_BANKS elements

```c
const int LOG_NUM BANKS = 4; // 16 banks
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
    if (thid <= d)
    {
        int a = s*(2*tid); int b = s*(2*tid+1)
        a += (a >> LOG_NUM BANKS); // insert pad word
        b += (b >> LOG_NUM BANKS); // insert pad word
        shared[a] += shared[b];
    }
}
```
Fixing Scan Bank Conflicts

- A full binary tree with 64 leaf nodes

<table>
<thead>
<tr>
<th>Leaf Nodes</th>
<th>Scale (s)</th>
<th>Thread addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td>0 2 4 6 8 10 12 14 17 19 21 23 25 27 29 31 34 36 38 40 42 44 46 48 51 53 55 57 59 61 63</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0 4 8 12 17 21 25 29 34 38 42 46 51 55 59 63</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0 8 17 25 34 42 51 59</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0 17 34 51</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0 34</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conflicts</th>
<th>Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0 2 4 6 8 10 12 14 1 3 5 7 9 11 13 15 2 4 6 8 10 12 14 0 3 5 7 9 11 13 15</td>
</tr>
<tr>
<td>None</td>
<td>0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15</td>
</tr>
<tr>
<td>None</td>
<td>0 8 1 9 2 10 3 11</td>
</tr>
<tr>
<td>None</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>None</td>
<td>0 2</td>
</tr>
</tbody>
</table>

- No more bank conflicts!
  - However, there are ~8 cycles overhead for addressing
    - For each s[a] += s[b] (8 cycles/iter. * 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal
Fixing Scan Bank Conflicts

- A full binary tree with 128 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)

<table>
<thead>
<tr>
<th>Scale (s)</th>
<th>Thread addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0  4  8  12  17  21  25  29  34  38  42  46  51  55  59  63  68  72  76  80  85  89  93  97  102  106  110  114  119  123  127  131</td>
</tr>
<tr>
<td>4</td>
<td>0  8  17  25  34  42  51  59  68  76  85  93  102  110  119  127</td>
</tr>
<tr>
<td>8</td>
<td>0  17  34  51  68  85  102  119</td>
</tr>
<tr>
<td>16</td>
<td>0  34  68  102</td>
</tr>
<tr>
<td>32</td>
<td>0  68</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
</tr>
</tbody>
</table>

- Padding inserted

<table>
<thead>
<tr>
<th>Conflicts</th>
<th>Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0  4  8  12  1  5  9  13  2  6  10  14  3  7  11  15  4  8  12  0  5  9  13  1  6  10  14  2  7  11  15  3</td>
</tr>
<tr>
<td>None</td>
<td>0  8  1  9  2  10  3  11  4  12  5  13  6  14  7  15</td>
</tr>
<tr>
<td>None</td>
<td>0  1  2  3  4  5  6  7</td>
</tr>
<tr>
<td>None</td>
<td>0  2  4  6</td>
</tr>
<tr>
<td>None</td>
<td>0  4</td>
</tr>
</tbody>
</table>

- No more bank conflicts!
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal
Fixing Scan Bank Conflicts

- A full binary tree with 512 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)

<table>
<thead>
<tr>
<th>Scale(s)</th>
<th>Thread addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 17 34 51 68 85 102 119 136 153 170 187 204 221 238 255 272 289 306 323 340 357 374 391 408 425 442 459 476 493 510 527</td>
</tr>
<tr>
<td>16</td>
<td>0 34 68 102 136 170 204 238 272 306 340 374 408 442 476 510</td>
</tr>
<tr>
<td>32</td>
<td>0 68 136 204 272 340 408 476</td>
</tr>
<tr>
<td>64</td>
<td>0 136 272 408</td>
</tr>
<tr>
<td>128</td>
<td>0 272</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
</tr>
</tbody>
</table>

Conflicts | Banks
---|---
None | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2-way | 2 4 6 8 10 12 14 0 2 4 6 8 10 12 14
2-way | 0 4 8 12 0 4 8 12
2-way | 0 8 0 8
None | 0

- Wait, we still have bank conflicts
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal
- But it does not pay of to optimize for the rest. Address calculations are getting too expensive
Summary

- Parallel Programming requires careful planning
  - of the branching behavior
  - of the memory access patterns
  - of the work efficiency

- Vector Reduction
  - branch efficient
  - bank efficient

- Scan Algorithm
  - based in Balanced Tree principle:
    bottom up, top down traversal
Shuffle: Tips and Tricks

Julien Demouth, NVIDIA
Glossary

Safer with cooperative thread groups!

- **Warp**
  - Implicitly synchronized group of threads (32 on current HW)

- **Warp ID** \((\text{warpid})\)
  - Identifier of the warp in a block: \(\text{threadIdx.x} / 32\)

- **Lane ID** \((\text{laneid})\)
  - Coordinate of the thread in a warp: \(\text{threadIdx.x} \mod 32\)
  - Special register (available from PTX): \%\text{laneid}
Shuffle (SHFL)

- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0
Variants

- 4 variants (idx, up, down, bfly):

Now: Use shuffle in cooperative thread groups!
Instruction (PTX)

Optional dst. predicate          Lane/offset/mask
shfl.mode.b32 d[|p], a, b, c;

Dst. register                    Src. register       Bound

Now: Use shuffle in cooperative thread groups!
Implement SHFL for 64b Numbers

```c
__device__ __inline__ double shfl(double x, int lane)
{
    // Split the double number into 2 32b registers.
    int lo, hi;
    asm volatile( "mov.b32 {%0,%1}, %2;" : "=r"(lo), "=r"(hi) : "d"(x));

    // Shuffle the two 32b registers.
    lo = __shfl(lo, lane);
    hi = __shfl(hi, lane);

    // Recreate the 64b number.
    asm volatile( "mov.b64 %0, {%1,%2};" : "=d(x)" : "r"(lo), "r"(hi));

    return x;
}
```

- **Generic SHFL:** [https://github.com/BryanCatanzaro/generics](https://github.com/BryanCatanzaro/generics)
Performance Experiment

- One element per thread
  
  \[
  \text{thread: } 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ \ldots
  \]

- Each thread takes its right neighbor
  
  \[
  \text{thread: } 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ \ldots
  \]
  
  \[
  \text{x: } \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quarter
Performance Experiment

- We run the following test on a K20

  ```
  T x = input[tidx];
  for(int i = 0 ; i < 4096 ; ++i)
      x = get_right_neighbor(x);
  output[tidx] = x;
  ```

- We launch 26 blocks of 1024 threads
  - On K20, we have 13 SMs
  - We need 2048 threads per SM to have 100% of occupancy

- We time different variants of that kernel
Performance Experiment

- Shared memory (SMEM)
  
  \[ \text{s} = \text{smem}[\text{threadIdx.x}] = \text{smem}[32^*\text{warpid} + ((\text{laneid}+1) \mod 32)]; \]
  
  \[ \_\_\text{syncthreads}(); \]

- Shuffle (SHFL)
  
  \[ x = \_\_\text{shfl}(x, (\text{laneid}+1) \mod 32); \]

- Shared memory without \_\_\text{syncthreads} + volatile (unsafe)
  
  \[ \_\_\text{shared} \_\_ \text{volatile } T^*\text{smem} = \ldots; \]
  
  \[ \text{s} = \text{smem}[\text{threadIdx.x}] = \text{smem}[32^*\text{warpid} + ((\text{laneid}+1) \mod 32)]; \]
Performance Experiment (fp32)

- **Execution Time (ms)**
  - SMEM
  - SMEM (unsafe)
  - SHFL

- **SMEM per Block (KB)**
  - SMEM
  - SMEM (unsafe)
  - SHFL
Performance Experiment (fp64)

**Execution Time (ms)**

- SMEM
- SMEM (unsafe)
- SHFL

**SMEM per Block (KB)**

- SMEM
- SMEM (unsafe)
- SHFL
Performance Experiment

- Always faster than shared memory

- Much safer than using no __syncthreads (and volatile)
  - And never slower

- Does not require shared memory
  - Useful when occupancy is limited by SMEM usage
Broadcast

Now: Use cooperative thread groups!

- All threads read from a single lane
  
  \[
  x = \_shfl(x, 0); \text{ // All the threads read } x \text{ from laneid } 0.
  \]

- More complex example

  ```c
  // All threads evaluate a predicate.
  int predicate = ...;

  // All threads vote.
  unsigned vote = \_ballot(predicate);

  // All threads get x from the "last" lane which evaluated the predicate to true.
  if(vote)
    x = \_shfl(x, \_bfind(vote));

  // \_bind(unsigned i): Find the most significant bit in a 32/64 number (PTX).
  \_bfind(&b, i) { asm volatile("bfind.u32 %0, %1;" : "=r"(b) : "r"(i)); }
  ```
Reduce

- Code

```c
// Threads want to reduce the value in x.
float x = ...;
#pragma unroll
for(int mask = WARP_SIZE / 2 ; mask > 0 ; mask >>= 1)
  x += __shfl_xor(x, mask);
// The x variable of laneid 0 contains the reduction.
```

- Performance
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times
Scan

- **Code**
  
  ```
  #pragma unroll
  for( int offset = 1 ; offset < 32 ; offset <<= 1 )
  {
    float y = __shfl_up(x, offset);
    if(laneid() >= offset)
      x += y;
  }
  ```

- **Performance**
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times
Scan

- Use the predicate from SHFL

```c
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <<= 1 )
{
    asm volatile("%" 
        " .reg .f32 r0;"
        " .reg .pred p;"
        " shfl.up.b32 r0 | p, %0, %1, 0x0;"
        " @p add.f32 r0, r0, %0;"
        " mov.f32 %0, r0;"
    
    "\" : "+f"(x) : "r"(offset));
}
```

- Use CUB:
  https://nvlabs.github.com/cub
Bitonic Sort

\[ x: \begin{array}{cccccccccccccc}
11 & 3 & 8 & 5 & 10 & 15 & 9 & 7 & 12 & 4 & 2 & 0 & 14 & 13 & 6 & 1 & \ldots
\end{array} \]

- \( \text{stride}=1 \)
  \[ \begin{array}{cccccccccccccccc}
3 & 11 & 8 & 5 & 10 & 15 & 9 & 7 & 4 & 12 & 2 & 0 & 13 & 14 & 6 & 1 & \ldots
\end{array} \]

- \( \text{stride}=2 \)
  \[ \begin{array}{cccccccccccccccc}
3 & 5 & 8 & 11 & 10 & 15 & 9 & 7 & 2 & 0 & 4 & 12 & 13 & 14 & 6 & 1 & \ldots
\end{array} \]

- \( \text{stride}=1 \)
  \[ \begin{array}{cccccccccccccccc}
3 & 5 & 8 & 11 & 15 & 10 & 9 & 7 & 0 & 2 & 4 & 12 & 14 & 13 & 6 & 1 & \ldots
\end{array} \]
Bitonic Sort

stride=4:
3 5 8 7 15 10 9 11 14 13 6 12 0 2 4 1 ...

stride=2:
3 5 8 7 9 10 15 11 14 13 6 12 4 2 0 1 ...

stride=1:
3 5 7 8 9 10 11 15 14 13 12 6 4 2 1 0 ...
Bitonic Sort

```c
int swap(int x, int mask, int dir)
{
    int y = __shfl_xor(x, mask);
    return x < y == dir ? y : x;
}
```

```c
x = swap(x, 0x01, bfe(laneid, 1) ^ bfe(laneid, 0)); // 2
x = swap(x, 0x02, bfe(laneid, 2) ^ bfe(laneid, 1)); // 4
x = swap(x, 0x01, bfe(laneid, 2) ^ bfe(laneid, 0));
```

```
Execution Time int32 (ms)
```

```
SMEM  SMEM (unsafe)  SHFL
```

```
SMEM per Block (KB)
```

```c
x = swap(x, 0x04, bfe(laneid, 3) ^ bfe(laneid, 2)); // 8
x = swap(x, 0x02, bfe(laneid, 3) ^ bfe(laneid, 1));
```

```
// int bfe(int i, int k): Extract k-th bit from i
```

```
// PTX: bfe dst, src, start, len (see p.81, ptx_isa_3.1)
```
**Transpose**

- When threads load or store arrays of structures, transposes enable fully coalesced memory operations.
- e.g. when loading, have the warp perform coalesced loads, then transpose to send the data to the appropriate thread.

Memory

<table>
<thead>
<tr>
<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>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

Registers

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>15</th>
<th>18</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td>20</td>
<td>23</td>
</tr>
</tbody>
</table>

\(m\) elements per thread

\(n\) threads in warp (8 for illustration only)
**Transpose**

- You can use SMEM to implement this transpose, or you can use SHFL.

- **Code:**
  
  http://github.com/bryancatanzaro/trove

- **Performance**
  
  - Launch 104 blocks of 256 threads
  - Run the transpose 4096 times
Array of Structures Access via Transpose

- Transpose speeds access to arrays of structures
- High-level interface: `coalesced_ptr<T>`
  - Just dereference like any pointer
  - Up to 6x faster than direct compiler generated access
Conclusion

- SHFL is available for SM $\geq$ SM 3.0
- It is always faster than “safe” shared memory
- It is never slower than “unsafe” shared memory
- It can be used in many different algorithms
WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0
Warp synchronous programming - NVIDIA Developer Forums
Jan 30, 2015 - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#volatile-qualifier regarding this: “If dynamically allocated shared memory of...

Is syncthreads required within a warp? - NVIDIA Developer Forums
https://devtalk.nvidia.com/default/topic/.../is-syncthreads-required-within-a-warp/ ▼
Nov 6, 2013 - cuda threads fence applied on share memory has the same effect only that it ... Warp synchronous programming is also safe across all current...

warp synchronous programming is a lie - Issue #167 - AccelerateHS ...
https://github.com/AccelerateHS/accelerate/issues/167 ▼
May 6, 2014 - Until at least CUDA 4, warp synchronous programming was the advertised and recommended way to get the best performance from a GPU.

cuda - What is warp-level-programming (racecheck) - Stack Overflow
stackoverflow.com/questions/1901826/what-is-warp-level-programming-racecheck ▼
Sep 25, 2013 - Its true that all processing is handled in warps. Warp level programming also called warp synchronous programming depends on this to ensure...

parallel processing - CUDA _syncthreads() usage within a warp ...
stackoverflow.com/questions/10205245/cuda-syncthreads-usage-within-a-warp ▼
Apr 18, 2012 - Updated with more information about using volatile. Presumably you want all threads ... On the face of it this means you can omit the _syncthreads(), a practice known as "warp-synchronous programming". However, there are...

cuda - Struggling with intuition regarding how warp-synchronous ...
stackoverflow.com/.../struggling-with-intuition-regarding-how-warp-synchronous-thr... ▼
Dec 4, 2013 - Let's look at the code in blocks, and answer your questions along the way. int sum ... Now we are finally getting into some warp-synchronous programming. This line of code depends on the fact that 32 threads are executing in...
CUDA WARP THREADING MODEL

NVIDIA GPU multiprocessors create, manage, schedule and execute threads in warps (32 parallel threads).

Threads in a warp may diverge and re-converge during execution.

Full efficiency may be realized when all 32 threads of a warp are converged.
WARP SYNCHRONOUS PROGRAMMING

Warp synchronous programming is a CUDA programming technique that leverages warp execution for efficient inter-thread communication.

- e.g. reduction, scan, aggregated atomic operation, etc.

CUDA C++ supports warp synchronous programming by providing warp synchronous built-in functions and cooperative group collectives.
EXAMPLE: SUM ACROSS A WARP

\[
\text{val} = \sum_{i=0}^{32} \text{input}[i]
\]

```c
val = input[lane_id];
val += __shfl_xor_sync(0xffffffff, val, 1);
val += __shfl_xor_sync(0xffffffff, val, 2);
val += __shfl_xor_sync(0xffffffff, val, 4);
val += __shfl_xor_sync(0xffffffff, val, 8);
val += __shfl_xor_sync(0xffffffff, val, 16);
```
HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

Make Sync Explicit

Thread re-convergence

- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.
HOW TO WRITE
WARP SYNCHRONOUS PROGRAMMING
Make Sync Explicit

Thread re-convergence
- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.

Data exchange between threads
- Use built-in functions to sync threads and exchange data in one step.
- When using shared memory, avoid data races between convergence points.

Reading and writing the same memory location by different threads may cause data races.
WARP SYNCHRONOUS BUILT-IN FUNCTIONS

Three Categories (New in CUDA 9.0)

Active-mask query: which threads in a warp are active
  • __activemask

Synchronized data exchange: exchange data between threads in warp
  • __all_sync, __any_sync, __uni_sync, __ballot_sync
  • __shfl_sync, __shfl_up_sync, __shfl_down_sync, __shfl_xor_sync
  • __match_any_sync, __match_all_sync

Threads synchronization: synchronize threads in a warp and provide a memory fence
  • __sincwarp
EXAMPLE: ALIGNED MEMORY COPY
  __activemask  __all_sync

// pick the optimal memory copy based on the alignment
__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
  unsigned mask = __activemask();

  if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
    return memcpy_aligned_16(tptr, sptr, size);

  if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
    return memcpy_aligned_8(tptr, sptr, size);

  ...
}
EXAMPLE: ALIGNED MEMORY COPY
__activemask  __all_sync

// pick the optimal memory copy based on the alignment
__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
    unsigned mask = __activemask();

    if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
        return memcpyAligned_16(tptr, sptr, size);

    if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
        return memcpyAligned_8(tptr, sptr, size);

    ...
}
EXAMPLE: ALIGNED MEMORY COPY

__activemask  __all_sync

// pick the optimal memory copy based on the alignment
__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
    unsigned mask = __activemask();
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
        return memcpy_aligned_16(tptr, sptr, size);
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
        return memcpy_aligned_8(tptr, sptr, size);
    ...
}

Find the active threads

Returns true when all threads in ‘mask’ have the same predicate value
EXAMPLE: SHUFFLE
__shfl_sync, __shfl_down_sync

Broadcast: all threads get the value of ‘x’ from lane id 0

\[ y = \_\_shfl\_sync(0xffffffff, x, 0); \]
EXAMPLE: SHUFFLE
__shfl_sync, __shfl_down_sync

Broadcast: all threads get the value of ‘x’ from lane id 0

\[ y = \_\_shfl\_sync(0xffffffff, x, 0); \]

Reduction:

```c
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(0xffffffff, val, offset);
```
EXAMPLE: DIVERGENT BRANCHES

All *_sync built-in functions can be used in divergent branches on Volta

```c
#define FULLMASK 0xffffffff
__device__ int get_warp_sum(int v) {
    for (int i = 1; i < 32; i = i*2)
        v += __shfl_xor_sync(FULLMASK, v, i);
    return v;
}
```
EXAMPLE: DIVERGENT BRANCHES
All *_sync built-in functions can be used in divergent branches on Volta

```c
#define FULLMASK 0xffffffff
__device__ int get_warp_sum(int v) {
    for (int i = 1; i < 32; i = i*2)
        v += __shfl_xor_sync(FULLMASK, v, i);
    return v;
}
```

Possible to write a library function that performs warp synchronous programming w/o requiring it to be called convergently.
EXAMPLE: REDUCTION VIA SHARED MEMORY

__syncwarp

Re-converge threads and perform memory fence

v += shm membr[tid+16]; __syncwarp();
shm membr[tid] = v; __syncwarp();
v += shm membr[tid+8]; __syncwarp();
shm membr[tid] = v; __syncwarp();
v += shm membr[tid+4]; __syncwarp();
shm membr[tid] = v; __syncwarp();
v += shm membr[tid+2]; __syncwarp();
shm membr[tid] = v; __syncwarp();
v += shm membr[tid+1]; __syncwarp();
shm membr[tid] = v;
BUT WHAT’S WRONG WITH THIS CODE?

```c
v += shmem[tid+16];
shmem[tid] = v;
v += shmem[tid+8];
shmem[tid] = v;
v += shmem[tid+4];
shmem[tid] = v;
v += shmem[tid+2];
shmem[tid] = v;
v += shmem[tid+1];
shmem[tid] = v;
```
Implicit warp synchronous programming builds upon two unreliable assumptions,
• implicit thread re-convergence points, and
• Implicit lock-step execution of threads in a warp.

Implicit warp synchronous programming is unsafe and unsupported.

Make warp synchronous programming safe by making synchronizations explicit.
IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```c
if (lane_id < 16)
    A;
else
    B;
assert(__activemask() == 0xffffffff);
```
IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```c
if (lane_id < 16)
    A;
else
    B;
assert(__activemask() == 0xffffffff); not guaranteed to be true
```

Solution

- Do not reply on implicit thread re-convergence
- Use warp synchronous built-in functions to ensure convergence
IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

```c
if (__activemask() == 0xffffffff) {
    assert(__activemask() == 0xffffffff);
}
```
IMPICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

if (__activemask() == 0xffffffff)
    assert(__activemask() == 0xffffffff); not guaranteed to be true

Solution

- Do not reply on implicit lock-step execution
- Use warp synchronous built-in functions to ensure convergence
IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 3

\[
\begin{align*}
\text{shmem}[\text{tid}] & += \text{shmem}[\text{tid}+16]; \\
\text{shmem}[\text{tid}] & += \text{shmem}[\text{tid}+8]; \\
\text{shmem}[\text{tid}] & += \text{shmem}[\text{tid}+4]; \\
\text{shmem}[\text{tid}] & += \text{shmem}[\text{tid}+2]; \\
\text{shmem}[\text{tid}] & += \text{shmem}[\text{tid}+1];
\end{align*}
\]
Implicit Lock-Step Execution

Unreliable Assumption 2

Example 3

```
shmem[tid] += shmem[tid+16];
shmem[tid] += shmem[tid+8];
shmem[tid] += shmem[tid+4];
shmem[tid] += shmem[tid+2];
shmem[tid] += shmem[tid+1];
```

Data race

Solution

```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v; __syncwarp();
v += shmem[tid+8]; __syncwarp();
shmem[tid] = v; __syncwarp();
v += shmem[tid+4]; __syncwarp();
shmem[tid] = v; __syncwarp();
v += shmem[tid+2]; __syncwarp();
shmem[tid] = v; __syncwarp();
v += shmem[tid+1]; __syncwarp();
shmem[tid] = v;
```

- Make sync explicit
LEGACY WARP-LEVEL BUILT-IN FUNCTIONS

Deprecated in CUDA 9.0

Legacy built-in functions

- __all(), __any(), __ballot(), __shfl(), __shfl_up(), __shfl_down(), __shfl_xor()

These legacy warp-level built-in functions can perform data exchange between the active threads in a warp.

They do not ensure which threads are active.

They are deprecated in CUDA 9.0 on all architectures.
### COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

**Example: warp aggregated atomic**

```c
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);

coalessced_group g = coalesced_threads();

int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;

int res;
if (g.thread_rank() == 0)
    res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;

int res;
if (rank == 0)
    res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```
WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

New warp synchronous built-in functions ensure reliable synchronizations.
New warp synchronous built-in functions can be used divergently on Volta.
Legacy warp built-in functions are deprecated.

Cooperative groups offers
- Higher-level abstraction of thread groups
- Four levels of thread grouping
- More scalable code and better software decomposition
BETTER COMPOSITION

Barrier synchronization hidden within functions

```c
__device__ int sum(int *x, int n)
{
    ...  
    __syncthreads();  
    ...  
    return total;
}

__global__ void parallel_kernel(float *x)
{
    ...  
    // Entire thread block must call sum  
    sum(x, n);  
}
```

All threads in thread block must arrive at this barrier.

Hidden constraint on caller due to implementation of sum.
BETTER COMPOSITION

Explicit cooperative interfaces

```cpp
__device__ int sum(thread_group g, int *x, int n)
{
    ...
    g.sync() // Participating thread group provided by caller.
    ...
    return total;
}

__global__ void parallel_kernel(...)
{
    ...
    // Entire thread block must call sum in `sum` is visible in code.
    sum(this_thread_block(), x, n);
    ...
}
```
FUTURE ROADMAP
Partition by label or predicate, more complex scopes

0 1 0 1 0 1 0 1 (Volta specific)

thread_group cta = this_thread_block();
thread_group g = partition(cta, cta.thread_rank() & 1);

Warp 32  Warp 32  Warp 32
0 0 0 0 0  0 0 0 0 0  1 1 1 1 1

thread_group g = tiled_partition(cta, 64);

At all scopes!
FUTURE ROADMAP

Library of collectives (sort, reduce, etc.)

```cpp
template <int BlockThreads>
__global__ int BlockReduce(float *d_in, ...)
{
    static_thread_block<BlockThreads> cta = this_thread_block();
    // Statically allocate shared reduction storage
    __shared__ reduce_storage<decltype(cta), float> groupReduce;

    // Compute the block-wide sum for thread-0
    float total = cooperative_groups::reduce(
        cta, d_in[cta.rank()], groupReduce);
}
```

On a simpler note:

```cpp
// Collective key-value sort, default allocator
cooperative_groups::sort(this_thread_block(), myValues, myKeys);
```
HONORABLE MENTION

The ones that didn’t make it into their own slide

_CG_DEBUG : Define to enable various runtime safety checks. This helps debug incorrect API usage, incorrect synchronization, or similar issues (Automatically turned on with -G).

Tools help detect incorrect warp-synchronization with the racecheck tool.

Match is a new Volta instruction that is able to return who in your warp has the same 32 or 64 bit value
Developers now have a flexible model for synchronization and communication between groups of threads.

Shipping in CUDA 9.0

Provides safety, composability, and high performance

Flexibility to synchronize at various architecture and program defined scopes.

Deploy everywhere from Kepler to Volta
Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA