Warp-synchronous programming with Cooperative Groups

Nov. 8, 2017

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
SIMT, SIMD: the best of both worlds?

SIMT model (NVIDIA GPUs)

- warp

Explicit SIMD model (AVX, AMD GPUs...)

- thread

- Common denominator: independent calculations
SIMT, SIMD: the best of both worlds?

SIMT model (NVIDIA GPUs)

- Common denominator: independent calculations

Explicit SIMD model (AVX, AMD GPUs...)

- Feature: automatic branch divergence management
- Feature: direct communication across SIMD lanes

Warp-synchronous programming: write explicit SIMD code in CUDA

Can we have both branch divergence and direct communication?
Example: sum 8 numbers in parallel

SIMT: CUDA C by the book

Registers
\[ t_0 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \]
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \]

Shared memory
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \]

Syncthreads

SIMD: Intel AVX

Registers
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \]

\[ \text{hadd} \]
\[ \text{hadd} \]
\[ \text{extract} \]
\[ \text{add} \]
\[ 3 \ 7 \ 11 \ 15 \]
\[ 10 \ 26 \]
\[ 10 \]
\[ 36 \]

\[ \rightarrow 4 \text{ arithmetic instructions} \]

+ 2 more times...
\[ \rightarrow 3 \text{ stores, 6 loads, 4 syncthreads, 3 adds} \]
+ address calculations!

- **SIMT: inter-thread communication through memory + block-level synchronization**
  - Overkill for threads of the same warp!
Agenda

- Introducing cooperative groups
  - API overview
  - Thread block tile
- Collective operations
  - Shuffle
  - Vote
  - Match
- Thread block tile examples
  - Reduction
  - Parallel prefix
  - Multi-precision addition
- Coalesced groups
  - Motivation
  - Example: stream compaction
Exposing the “warp” level

Before CUDA 9.0, no level between Thread and Thread Block in programming model

- Warp-synchronous programming: arcane art relying on undefined behavior
Before CUDA 9.0, no level between Thread and Thread Block in programming model

- *Warp-synchronous programming*: arcane art relying on undefined behavior

CUDA 9.0 Cooperative Groups: let programmers define extra levels

- Fully exposed to compiler and architecture: safe, well-defined behavior
- Simple C++ interface: warp-synchronous programming for the masses!
The cooperative group API

- No magic: cooperative groups is a device-side C++ library
  - You can the read the code: cuda/include/cooperative_groups.h in CUDA Toolkit 9.0
- Supports group sizes all the way from single-thread to multi-grid
The cooperative group API

<table>
<thead>
<tr>
<th>C++ library</th>
<th>Thread group</th>
<th>Thread block tile group</th>
<th>Coalesced group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-grid group</td>
<td>Grid group</td>
<td>Thread block group</td>
<td></td>
</tr>
<tr>
<td>cudaCG API</td>
<td>Good old CUDA C</td>
<td></td>
<td>Warp-synchronous primitives</td>
</tr>
</tbody>
</table>

- No magic: cooperative groups is a device-side C++ library
  - You can read the code: cuda/include/cooperative_groups.h in CUDA Toolkit 9.0
- Supports group sizes all the way from single-thread to multi-grid
  - In this tutorial, focus on warp-sized groups
Common cooperative groups features

- Base class for all groups: `thread_group`
- Specific thread group classes derive from `thread_group`

In namespace `cooperative_groups`

```cpp
class thread_group {
public:
  __device__ unsigned int size() const;
  __device__ unsigned int thread_rank() const;
  __device__ void sync() const;
};
```

- Number of threads in group
- Identifier of this thread within group
- Synchronization barrier: like `__syncthreads` within a group
Some simple groups

- Single-thread group
  - thread_group myself = this_thread();
  - Creates groups of size 1, all threads have rank 0, sync is a no-op

- Thread block group
  - thread_block myblock = this_thread_block();
  - You could have written class thread_block in your sleep:

```cpp
class thread_block : public thread_group
{
  public:
    __device__ unsigned int size() const {
      return blockDim.x * blockDim.y * blockDim.z;
    }

    __device__ unsigned int thread_rank() const {
      return (threadIdx.z * blockDim.y * blockDim.x) +
             (threadIdx.y * blockDim.x) +
             threadIdx.x;
    }

    __device__ void sync() const { __syncthreads(); }

    // Additional functionality exposed by the group
    __device__ dim3 group_index() const { return blockIdx; }
    __device__ dim3 thread_index() const { return threadIdx; }
}
```

Thread group
Thread block group
Good old CUDA C
Thread block tile

- Static partition of a group
  
  \[
  \text{thread\_block\_tile}<8> \text{ tile8} = \text{tiled\_partition}\langle8\rangle\text{(this\_thread\_block())};
  \]

  - Supported tile sizes now: power-of-2, \(\leq\) warp size: 1, 2, 4, 8, 16 or 32
    - All threads of a given tile belong to the same warp

- All threads participate: no gap in partitioning

Also: \(\text{thread\_group g} = \text{tiled\_partition}\text{(this\_thread\_block(), 8);}\)
Agenda

- Introducing cooperative groups
  - API overview
  - Thread block tile
- Collective operations
  - Shuffle
  - Vote
  - Match
- Thread block tile examples
  - Reduction
  - Parallel prefix
  - Multi-precision addition
- Coalesced groups
  - Motivation
  - Example: stream compaction
Thread block tile: collective operations

Enable direct communication between threads of a `thread_block_tile`.

```cpp
template <unsigned int Size>
class thread_block_tile : public thread_group
{
    public:
        __device__ void sync() const;
        __device__ unsigned int thread_rank() const;
        __device__ unsigned int size() const;

    // Shuffle collectives
        __device__ int shfl(int var, int srcRank) const;
        __device__ int shfl_down(int var, unsigned int delta) const;
        __device__ int shfl_up(int var, unsigned int delta) const;
        __device__ int shfl_xor(int var, unsigned int laneMask);

    // Vote collectives
        __device__ int any(int predicate) const;
        __device__ int all(int predicate) const;
        __device__ unsigned int ballot(int predicate);

    // Match collectives
        __device__ unsigned int match_any(int val);
        __device__ unsigned int match_all(int val, int &pred);
};
```
Shuffle collectives: generic shuffle

\[ \text{g.shfl}(v, i) \] returns value of \( v \) of thread \( i \) in the group

- Use cases
  - Arbitrary permutation
  - Up to 32 concurrent lookups in a 32-entry table
  - Broadcast value of a given thread \( i \) to all threads

Example with tile size 16

\[
\begin{array}{cccccccccccccccc}
   & t0 & t1 & t2 & & & & & t15 \\
\hline
  i & 1 & 2 & 3 & 0 & 9 & 4 & 7 & 1 & 2 & 1 & 5 & 4 & 1 & 9 & 12 & 10 \\
  v & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\end{array}
\]

\[ \text{g.shfl}(v, i) \]
Shuffle collectives: specialized shuffles

- \( g.\text{shfl}_\text{up}(v, i) \approx g.\text{shfl}(v, \text{rank} - i) \),
- \( g.\text{shfl}_\text{down}(v, i) \approx g.\text{shfl}(v, \text{rank} + i) \)

Index is relative to the current lane

- Use: neighbor communication, shift

- \( g.\text{shfl}_\text{xor}(v, i) \approx g.\text{shfl}(v, \text{rank} ^ \wedge i) \)

- Use: exchange data pairwise: “butterfly”

Old value of \( v \): not zero!

\[ \begin{align*}
\text{shfl}_\text{up}(v, 4): & \quad 0123456789101112131415 \\
\text{shfl}_\text{xor}(v, 2): & \quad 23016745101189\ldots
\end{align*} \]
Warp vote collectives

- `bool p2 = g.all(p1)`
  - horizontal AND between predicates p1
  - Returns true when all inputs are true

- `bool p2 = g.any(p1)`
  - OR between all p1
  - Returns true if any input is true

- `uint n = g.ballot(p)`
  - Set bit i of integer n to value of p for thread i
  - i.e. get bit mask as an integer
  - Least significant bit first: read right-to-left!

Use: take control decisions for the whole warp group
Match collectives

- New in Volta (requires Compute Capability ≥7.0)
- Returns set of threads that have the same value, as a bit mask
  - `uint m = g.match_any(key)`
  - `uint m = g.match_all(key, &pred)`

<table>
<thead>
<tr>
<th>key</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>42</td>
<td>17</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

(In binary, LSB first)

- `match_any`
- `match_all`

Use: conflict/sharing/divergence detection, binning
  - Powerful but low-level primitive
Threads in a warp can diverge and converge **anywhere** at **any time**
- Sync **waits** for other threads within group (may or may not converge threads)
- If one thread calls sync, all threads of the group need to call sync
  - Should be the same call to sync on pre-Volta archs

```c
if(g.thread_rank < 5){
    if(a[0] == 17) {
        g.sync();
    }
} else {
    ... 
}
g.sync();
```

Same condition for all threads in the group

**Correct**

Collective operations implicitly sync
- Same rules apply

**Correct**

**Only for CC ≥ 7.0**
Agenda

- Introducing cooperative groups
  - API overview
  - Thread block tile
- Collective operations
  - Shuffle
  - Vote
  - Match
- Thread block tile examples
  - Reduction
  - Parallel prefix
  - Multi-precision addition
- Coalesced groups
  - Motivation
  - Example: stream compaction
Example 1: reduction + broadcast

Naive algorithm

Step 0

\[
\begin{array}{cccccccc}
\Sigma_0-1 & \Sigma_2-3 & \Sigma_4-5 & \Sigma_6-7 \\
\end{array}
\]

\[a[2*i] \leftarrow a[2*i] + a[2*i+1]\]

\[a[4*i] \leftarrow a[4*i] + a[4*i+2]\]

\[a[8*i] \leftarrow a[8*i] + a[8*i+4]\]

\[a[i] \leftarrow a[0]\]

\[\Sigma_{i-j} \text{ is shorthand for } \sum_{k=i}^{j} a_k\]

Let's rewrite it using shuffles
Example 1: reduction + broadcast

Using butterfly shuffle

\[
\begin{align*}
\text{Step 0} & : & t_0 & \oplus & t_1 & \oplus & t_2 & \oplus & t_3 & \oplus & t_4 & \oplus & t_5 & \oplus & t_6 & \oplus & t_7 \\
\Sigma_0-1 & \Sigma_0-1 & \Sigma_2-3 & \Sigma_2-3 & \Sigma_4-5 & \Sigma_4-5 & \Sigma_6-7 & \Sigma_6-7 \\
\text{Step 1} & : & t_0 & \oplus & t_1 & \oplus & t_2 & \oplus & t_3 & \oplus & t_4 & \oplus & t_5 & \oplus & t_6 & \oplus & t_7 \\
\Sigma_0-3 & \Sigma_0-3 & \Sigma_0-3 & \Sigma_0-3 & \Sigma_4-7 & \Sigma_4-7 & \Sigma_4-7 & \Sigma_4-7 \\
\text{Step 2} & : & t_0 & \oplus & t_1 & \oplus & t_2 & \oplus & t_3 & \oplus & t_4 & \oplus & t_5 & \oplus & t_6 & \oplus & t_7 \\
\Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 & \Sigma_0-7 \\
\end{align*}
\]

\[
ai += \text{g.shfl_xor}(ai, 1); \\
ai += \text{g.shfl_xor}(ai, 2); \\
ai += \text{g.shfl_xor}(ai, 4); \\
\]

\(ai\) is a register: no memory access!
Example 2: parallel prefix

Kogge-Stone algorithm

Step 0

\[\begin{align*}
  & s[i] \leftarrow a[i] \\
  & \text{if } i \geq 1 \text{ then} \\
  & \quad s[i] \leftarrow s[i-1] + s[i] \\
\end{align*}\]

Step 1

\[\begin{align*}
  & \text{if } i \geq 2 \text{ then} \\
  & \quad s[i] \leftarrow s[i-2] + s[i] \\
\end{align*}\]

Step 2

\[\begin{align*}
  & \text{if } i \geq 4 \text{ then} \\
  & \quad s[i] \leftarrow s[i-4] + s[i] \\
\end{align*}\]

Step d: if \( i \geq 2^d \) then
\[s[i] \leftarrow s[i-2^d] + s[i]\]
Example 2: parallel prefix

Using warp-synchronous programming

```
s = a;
n = g.shfl_up(s, 1);
if(g.thread_rank() >= 1)
    s += n;
n = g.shfl_up(s, 2);
if(g.thread_rank() >= 2)
    s += n;
n = g.shfl_up(s, 4);
if(g.thread_rank() >= 4)
    s += n;
for(d = 1; d < 8; d *= 2) {
    n = g.shfl_up(s, d);
    if(g.thread_rank() >= d)
        s += n;
}
g.shfl_up does implicit sync: must stay outside divergent if!
```
Example 3: multi-precision addition

Add two 1024-bit integers together

- Represent big integers as vectors of 32×32-bit
  - A warp works on a single big integer
  - Each thread works on a 32-bit digit

- First step: add vector elements in parallel and recover carries

\[
\begin{array}{cccc}
\text{a7} & \text{a6} & \text{a5} & \text{a4} \\
\text{a3} & \text{a2} & \text{a1} & \text{a0} \\
\text{b7} & \text{b6} & \text{b5} & \text{b4} \\
\text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\hline
\text{r7} & \text{r6} & \text{r5} & \text{r4} \\
\text{r3} & \text{r2} & \text{r1} & \text{r0} \\
\end{array}
\]

\[
\text{t0} \quad \text{t1} \quad \text{t2} \quad \text{t31}
\]

```
uint32_t a = A[tid], b = B[tid], r, c;

r = a + b;  // Sum

c = r < a;  // Get carry
```
Second step: propagate carries

- This is a parallel prefix operation
  - We can do it in \( \log(n) \) steps
- But in most cases, one step will be enough
  - Loop until all carries are propagated

```c
uint32_t a = A[tid],
    b = B[tid], r, c;

r = a + b;  // Sum
r = r + c;  // Sum carry
```

```c
while(g.any(c)) {
  c = g.shfl_up(c, 1);  // Move left
  if(g.thread_rank() == 0) c = 0;
  r = r + c;  // Sum carry
}
```

```c
R[tid] = r;
```
We have prefix-parallel hardware for propagating carries: the adder!

- Ballot gathers all carries in one integer
- + propagates carries in one step
- And a few bitwise ops...

```c
uint32_t a = A[tid],
        b = B[tid], r, c;

r = a + b;    // Sum
r = r < a;    // Get carry
uint32_t gen = g.ballot(c);  // All generated carries
uint32_t prop = g.ballot(r == 0xffffffff);  // Propagations

gen = (gen + (prop | gen)) & ~prop;    // Propagate carries
r += (gen >> g.thread_rank()) & 1;    // Unpack an add carry
R[tid] = r;
```

e.g. in decimal:

<table>
<thead>
<tr>
<th></th>
<th>9</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>9</th>
<th>9</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c/gen</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prop/gen</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gen+prop</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&amp;~prop</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

→ [9 1 2 4 9 9 1 4 9 9]
Agenda

- Introducing cooperative groups
  - API overview
  - Thread block tile
- Collective operations
  - Shuffle
  - Vote
  - Match
- Thread block tile examples
  - Reduction
  - Parallel prefix
  - Multi-precision addition
- Coalesced groups
  - Motivation
  - Example: stream compaction
What about divergence management?

SIMT model (NVIDIA GPUs)
- Common denominator: independent calculations
- Feature: automatic branch divergence management
- Thread block tiles enable direct inter-lane communication
  - What about branch divergence?

Explicit SIMD model (AVX, AMD GPUs...)
- Feature: direct communication across SIMD lanes
Coalesced group

- Limitations of thread block tile
  - Regular partitioning only
  - Requires all threads to be active

- Coalesced group
  - Sparse subset of a warp made of all active threads
  - Dynamic: set at runtime
  - Supports thread divergence: can be nested

```c
if(condition) {
    coalesced_group g = coalesced_threads();
}
```

condition: 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0

g.thread_rank: 0 1 2 3 4 5 6 7

g.size() = 8
Collective operations on coalesced groups

- Support full assortment of shuffle, vote, match!
  - All indexing is based on computed thread rank
  - e.g. \texttt{g.shfl(v, i)}:
Collective operations on coalesced groups

- Support full assortment of shuffle, vote, match!
  - All indexing is based on computed thread rank
  - e.g. g.shfl(v, i):

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

<table>
<thead>
<tr>
<th>v</th>
<th>2</th>
<th>5 6</th>
<th>8</th>
<th>13</th>
<th>17 18</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>6</td>
<td>6 13</td>
<td>5</td>
<td>18</td>
<td>2 6</td>
<td>23</td>
</tr>
</tbody>
</table>

- You can just ignore inactive threads
- Beware of performance impact of thread rank remapping!
Collective operations on coalesced groups

- Support full assortment of shuffle, vote, match!
  - All indexing is based on computed thread rank
  - e.g. `g.shfl(v, i):

```
thread_rank
  0 1 2 3 4 5 6 7
i   2 2 4 1 6 0 2 7
v   2 5 6 8 1 3 1 7 1 8 2 3
result
   6 6 1 3 5 1 8 2 6 2 3
```

- You can just ignore inactive threads
- Beware of performance impact of thread rank remapping!
Filter out zero entries in a stream

- How I would like to implement it:

```c
__device__ int stream_compact(thread_block_tile<32> warp, 
    float input[], float output[], int n) {
    int j = 0;
    for(int i = warp.thread_rank(); i < n; i += warp.size()) {
        float x = input[i];
        if(x != 0.f) {
            coalesced_group g = coalesced_threads();
            output[j + g.thread_rank()] = x;
        }
        j += g.size();
    }
    return j;
}
```

- Beside the g scope issue, this code has a logic flaw!
  - Can you spot it?

Obviously invalid: g is out of scope and never existed for some threads!
Issue: intra-warp race condition

- Threads in warp can diverge at any time
  - One coalesced_group for each diverged path
  - All overwriting the same elements!

```c
__device__ int stream_compact(thread_block_tile<32> warp, float input[], float output[], int n) {
    int j = 0;
    for(int i = warp.thread_rank(); i < n; i += warp.size()) {
        float x = input[i];
        if(x != 0.f) {
            coalesced_group g = coalesced_threads();
            output[j + g.thread_rank()] = x;
        }
        j += g.size();
    }
    return j;
}
```

- Do not do that! *It might even work.*
The one and only coalesced_group example

Falling back to NVIDIA's official example...

- Aggregate multiple atomic increments to the same pointer from multiple threads

```c
__device__ int atomicAggInc(int *ptr)
{
    cg::coalesced_group g = coalesced_threads();
    int prev;
    // elect the first active thread to perform atomic add
    if (g.thread_rank() == 0) {
        prev = atomicAdd(ptr, g.size());
    }
    // broadcast previous value within the warp
    // and add each active thread’s rank to it
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

- Bottom line: use atomics to avoid race conditions:
  - Between different warps
  - Between (diverged) threads of the same warp

Implicit sync
Warp-synchronous code in functions

- Function using blocks or block tiles
  - Must be called by all threads of the block
  - Pass group as explicit parameter to expose this requirement
    
    ```c
    __device__ void foo(thread_block_tile<32> g, ...);
    ```
  - Convention that makes mistake of divergent call “harder to make”
  - Still not foolproof: no compiler check

- Function using coalesced group: freely composable!
  
  ```c
  __device__ void bar(...) {
    coalesced_group g = coalesced_threads();
  }
  ```
  - Same interface as a regular device function
    - Use of coalesced group is an implementation detail
    - Key improvement: enables composability of library code
Current limitations of cooperative groups

- Performance or flexibility, not both
  - Coalesced groups address code composability issues 😊
  - Warp-level collective operations on coalesced groups: currently much slower than on tiled partitions ☹
  - Future hardware support for coalesced groups?
  - Support for reactivating (context-switching) threads?

- Only support regular tiling, and subset of active threads
  - Hard to communicate data between different conditional paths
  - No wrapper over match collectives yet
  - Irregular partitions are one the roadmap!
    - e.g. auto irregular_partition = coalesced_threads().partition(key);

- Good news: none of these are fundamental problems
Takeaway

- Yet another level in the CUDA Grid hierarchy!
  - Blocks in grid: independent tasks, no synchronization
  - Thread groups in block: can communicate through shared memory
  - Threads in group: can communicate through registers

- Warp-synchronous programming is finally properly exposed in CUDA 9
  - Potential to write very efficient code: e.g. Halloc, CUB…
  - Not just for ninja programmers any more!
References

- Yuan Lin, Kyrylo Perelygin. *A Robust and Scalable CUDA Parallel Programming Model*. GTC 2017 presentation
  http://on-demand-gtc.gputechconf.com/gtc-quicklink/eLDK0P8

- Mark Harris, Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. ||∀ blog
  https://devblogs.nvidia.com/parallelforall/cooperative-groups/

- The Source!
  cuda-9.0/include/cooperative_groups.h