

---

# Warp-synchronous programming with Cooperative Groups

---

January 2020

Caroline Collange

Inria Rennes – Bretagne Atlantique

<https://team.inria.fr/pacap/members/collange/>

caroline.collange@inria.fr

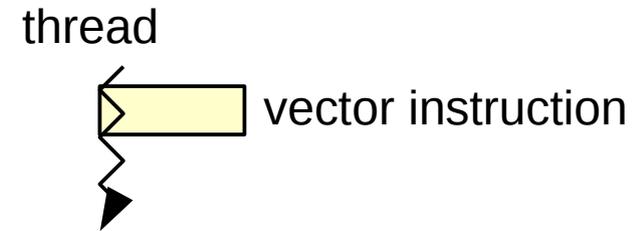


# SIMT, SIMD: common points

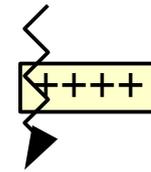
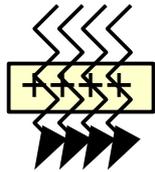
SIMT model (*NVIDIA GPUs*)



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

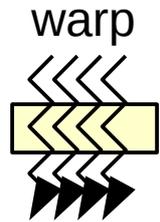


- Common denominator: independent calculations

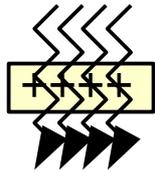


# SIMT, SIMD: differences

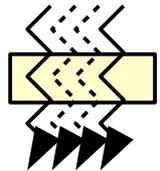
SIMT model (*NVIDIA GPUs*)



- Common denominator: independent calculations

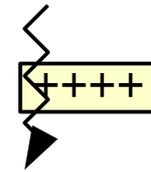
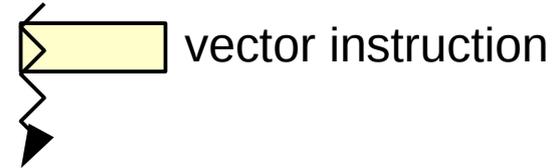


- Feature: automatic branch divergence management



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

thread



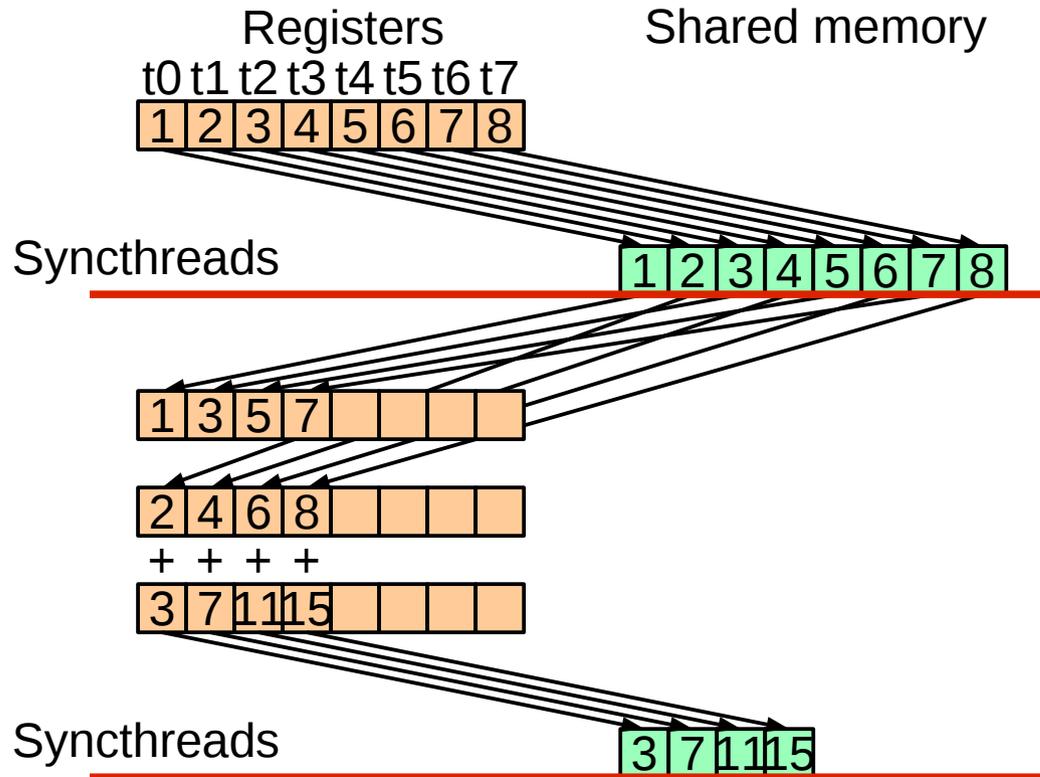
- Feature: direct communication across SIMD lanes



- *Warp-synchronous programming*: write explicit SIMD code in CUDA
  - Can we have both branch divergence **and** direct communication?

# Example: sum 8 numbers in parallel

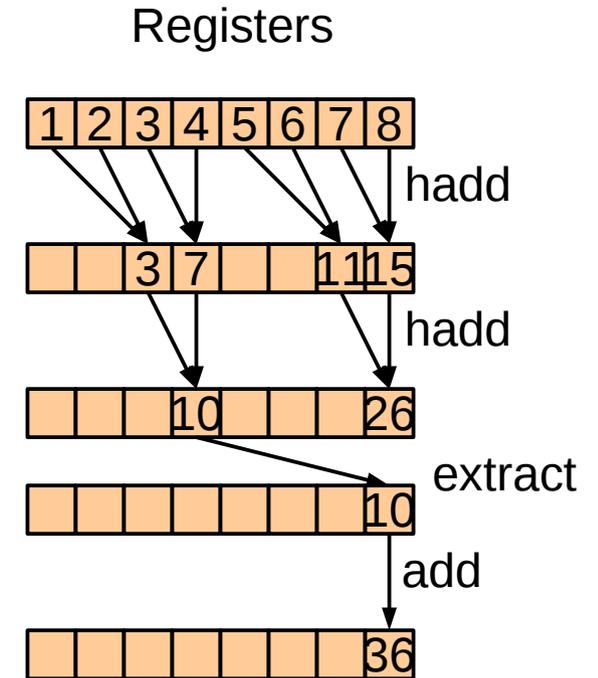
SIMT: CUDA C by the book



+ 2 more times...

→ 3 stores, 6 loads, 4 syncthreads, 3 adds  
+ address calculations!

SIMD: Intel AVX



→ 4 arithmetic instructions

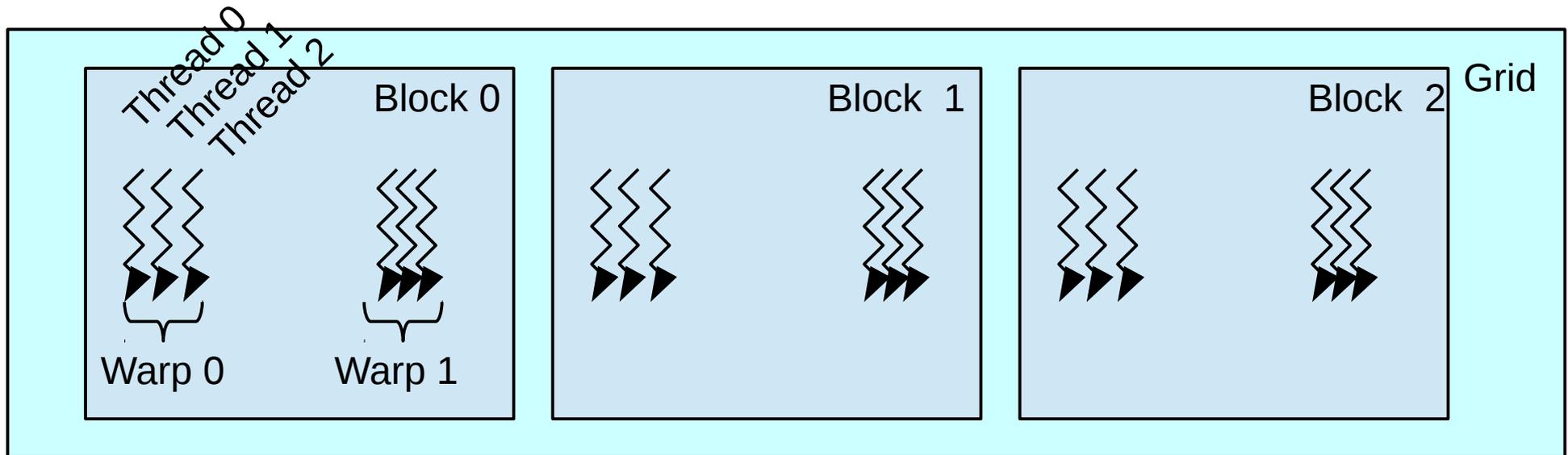
- SIMT: inter-thread communication through memory + block-level synchronization
  - ◆ Overkill for threads of the same warp!

# Agenda

---

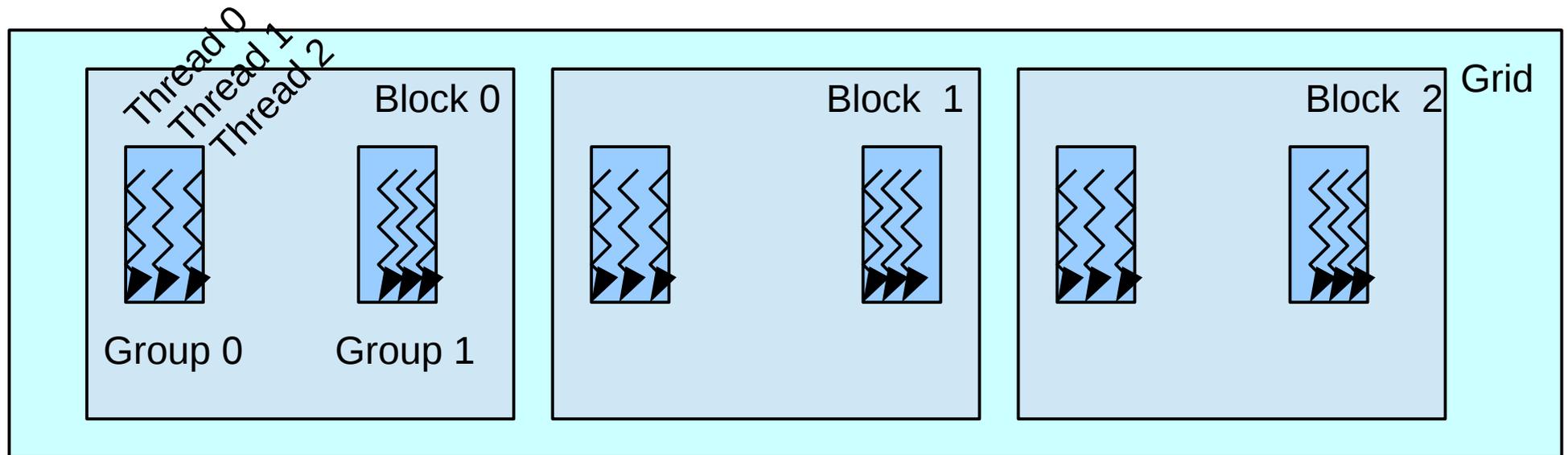
- Introducing cooperative groups
  - ◆ API overview
  - ◆ Thread block tile
- Collective operations
  - ◆ Shuffle
  - ◆ Vote
  - ◆ Match
- Thread block tile examples
  - ◆ Reduction
  - ◆ Parallel prefix
  - ◆ Multi-precision addition
- Coalesced groups
  - ◆ Motivation
  - ◆ Example: stream compaction

# Exposing the “warp” level



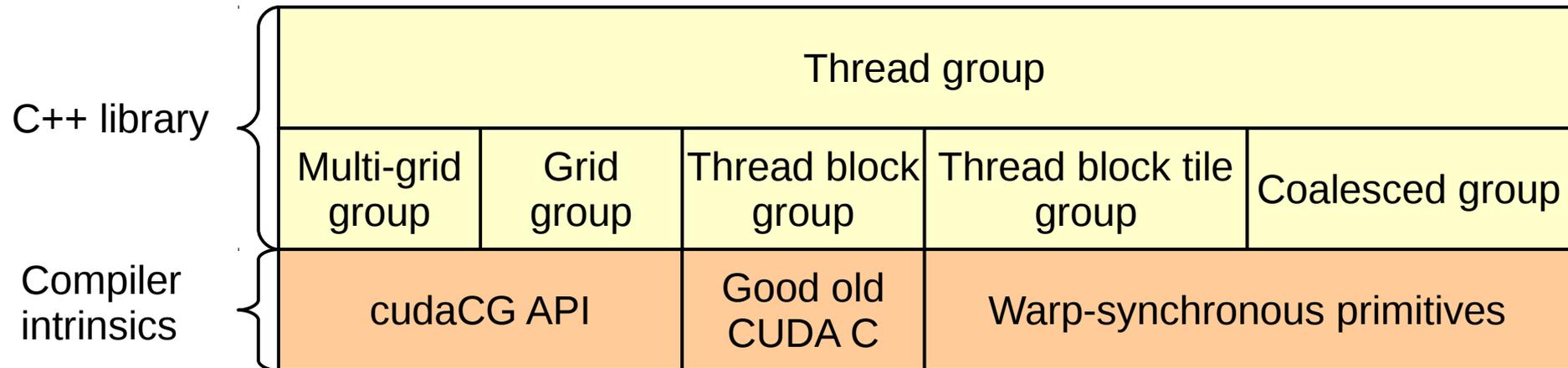
- Before CUDA 9.0, no level between Thread and Thread Block in programming model
  - ◆ *Warp-synchronous programming*: arcane art relying on undefined behavior

# Exposing the “warp” level



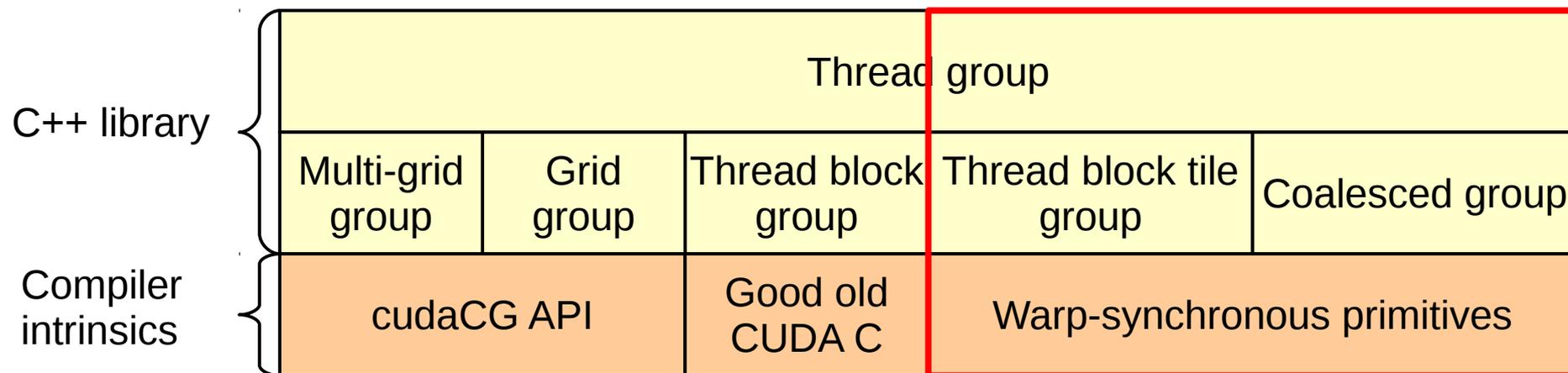
- Before CUDA 9.0, no level between Thread and Thread Block in programming model
  - ◆ *Warp-synchronous programming*: arcane art relying on undefined behavior
- CUDA 9.0 Cooperative Groups: let programmers define extra levels
  - ◆ Fully exposed to compiler and architecture: safe, well-defined behavior
  - ◆ Simple C++ interface

# The cooperative group API



- 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 (may change without notice!)
- Supports group sizes all the way from single-thread to multi-grid

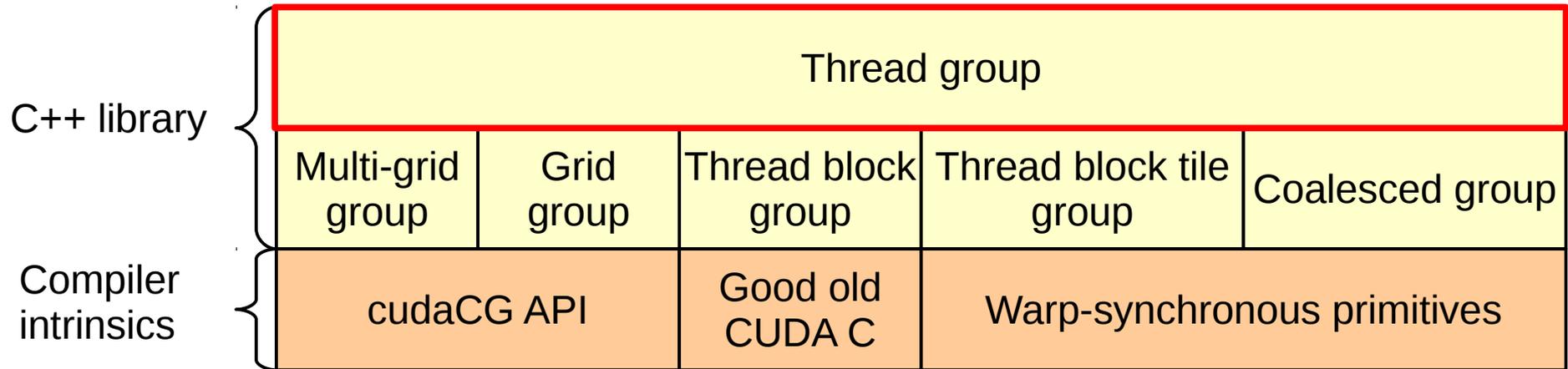
# The cooperative group API



- 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 (may change without notice!)
- Supports group sizes all the way from single-thread to multi-grid
  - ◆ In this lecture, 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`

```
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`:

```
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

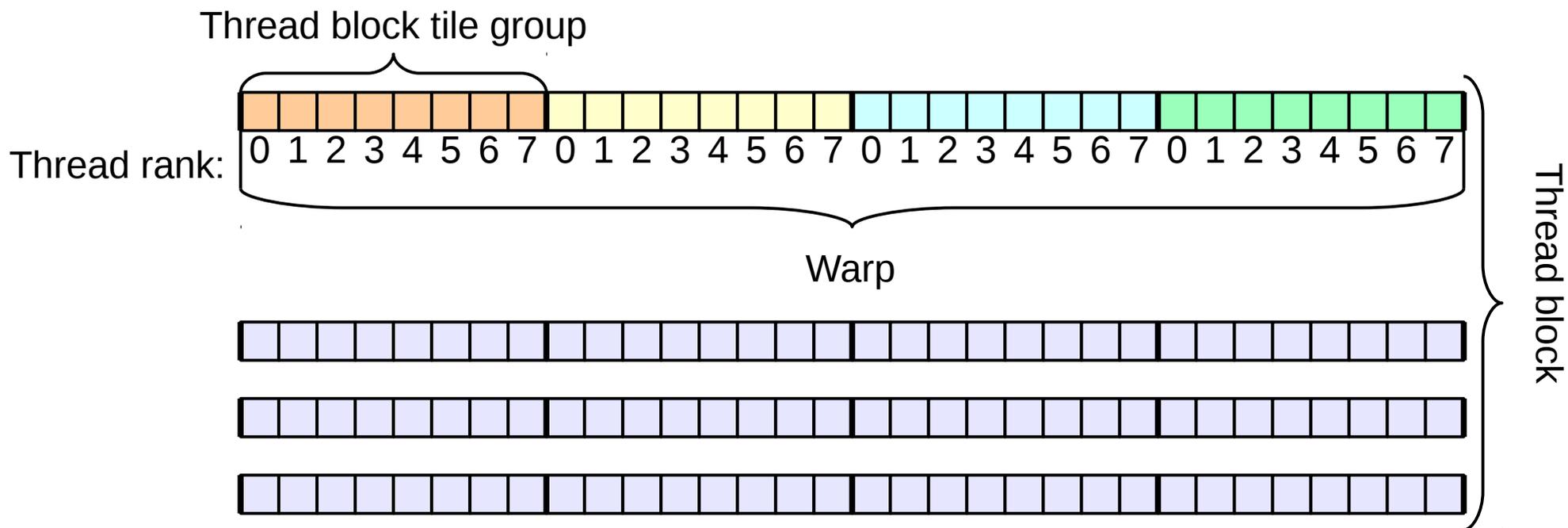
Good old  
CUDA C

# Thread block tile

- Static partition of a group

```
thread_block_tile<8> tile8 = tiled_partition<8>(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: `thread_group g = tiled_partition(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`

```
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);
};
```

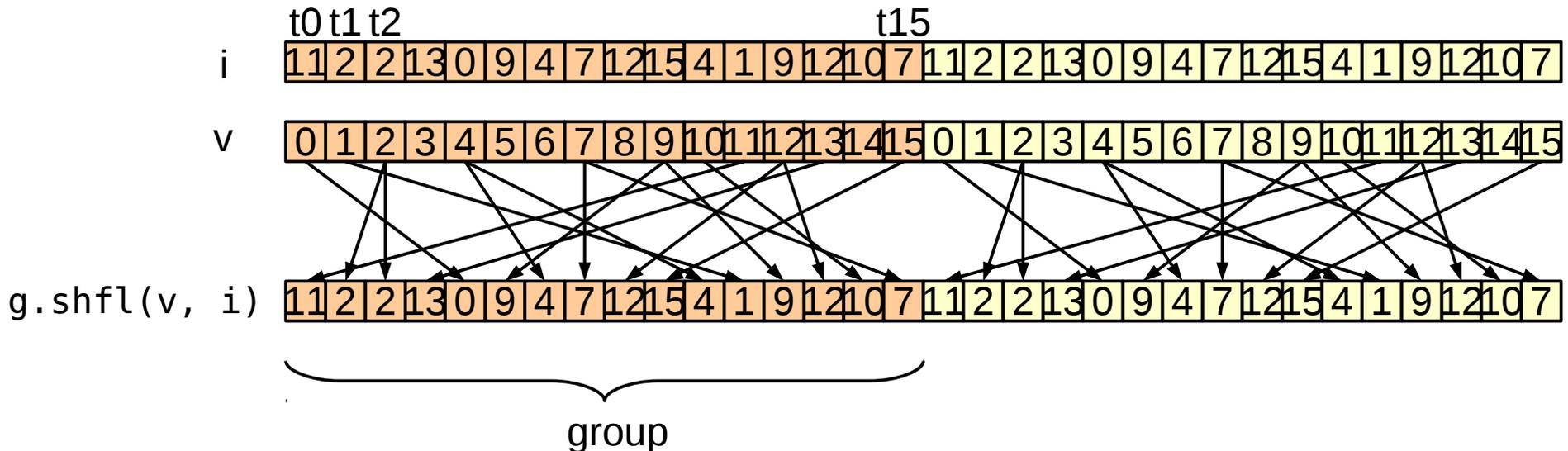
# Shuffle collectives: generic shuffle

`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 : like `v[i]`
- ◆ Broadcast value of a given thread `i` to all threads, when `i` is fixed

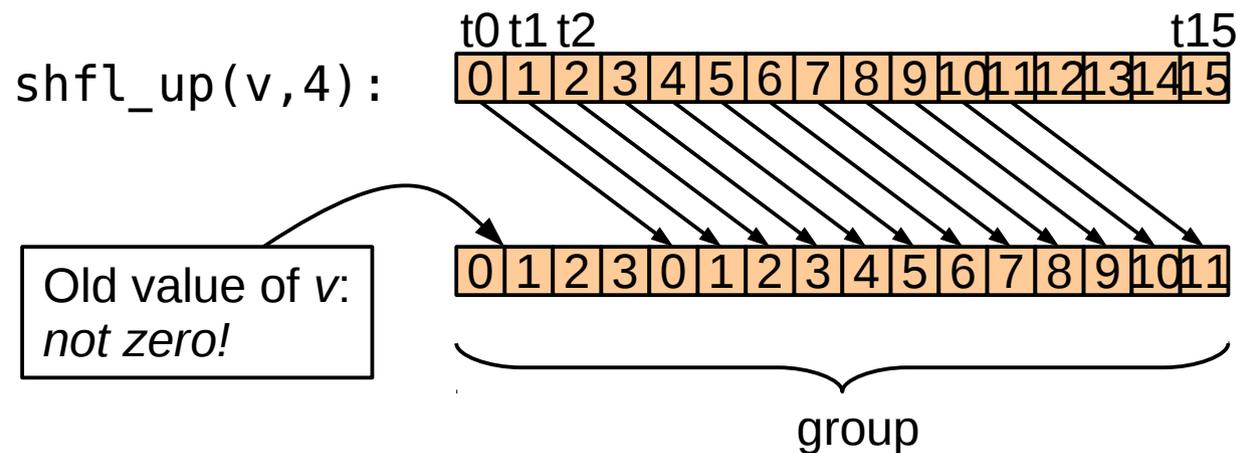
Example with tile size 16



# Shuffle collectives: specialized shuffles

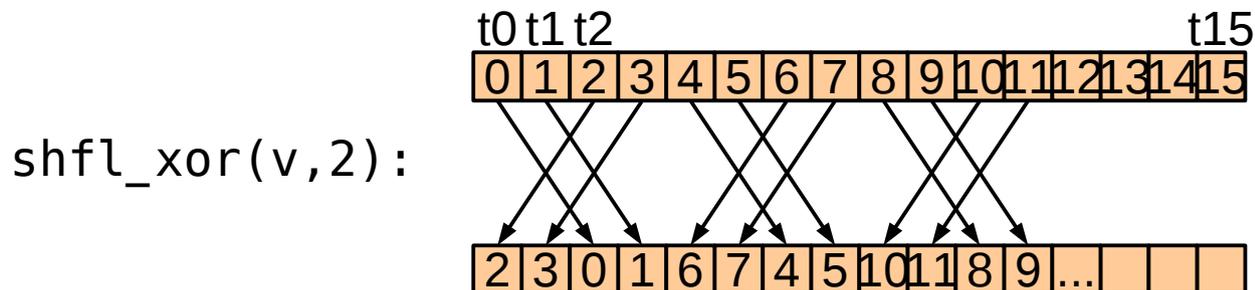
- $g.shfl\_up(v, i) \approx g.shfl(v, rank-i)$ ,  
 $g.shfl\_down(v, i) \approx g.shfl(v, rank+i)$   
Index is relative to the current lane

- ◆ Use: neighbor communication, shift



- $g.shfl\_xor(v, i) \approx g.shfl(v, rank \wedge i)$

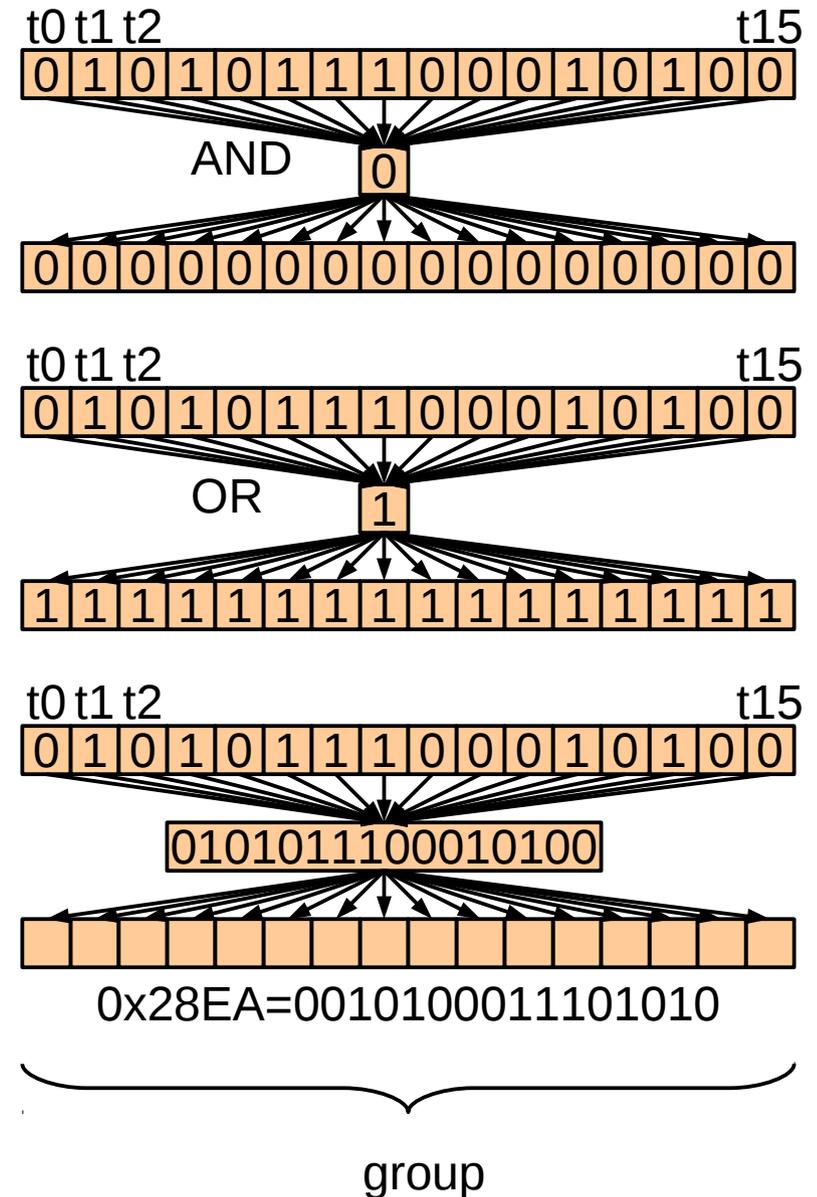
- ◆ Use: exchange data pairwise: “butterfly”



# Warp vote collectives

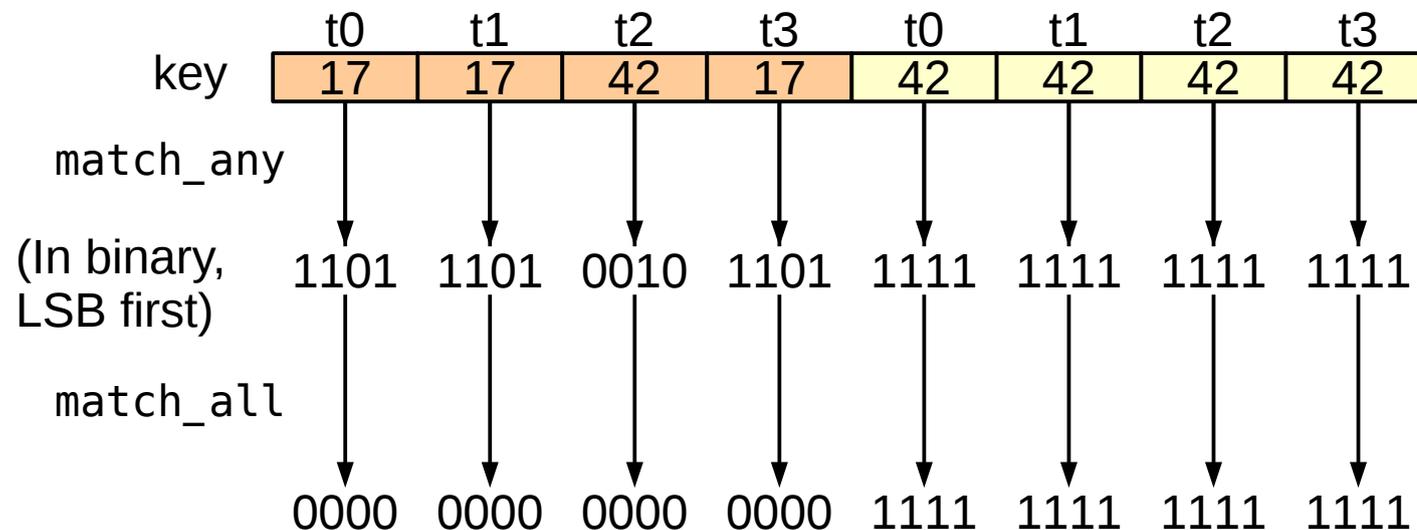
- `bool p2 = g.all(p1)`  
horizontal AND between predicates p1
  - ◆ Returns true when **all** inputs are true
- `bool p2 = g.any(p1)`  
OR between all p1
  - ◆ Returns true if **any** input is true
- `uint n = g.ballot(p)`  
Set bit *i* of integer *n*  
to value of *p* for thread *i*  
i.e. get bit mask as an integer
  - ◆ Least significant bit first:  
read right-to-left!

Use: take control decisions for the whole warp



# Match collectives

- New in Volta (requires Compute Capability  $\geq 7.0$ )
- 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)`



- Use: conflict/sharing/divergence detection, binning
  - ◆ Powerful but low-level primitive

# Group synchronization and divergence

- 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

```
if(g.thread_rank < 5){  
    ...  
}  
else {  
    ...  
}  
g.sync();
```

Correct

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

Correct

Same condition  
for all threads in the group

```
if(g.thread_rank() < 5){  
    g.sync();  
}  
else {  
    g.sync();  
}
```

Only for CC ≥ 7.0

- Collective operations implicitly sync
  - ◆ Same rules apply

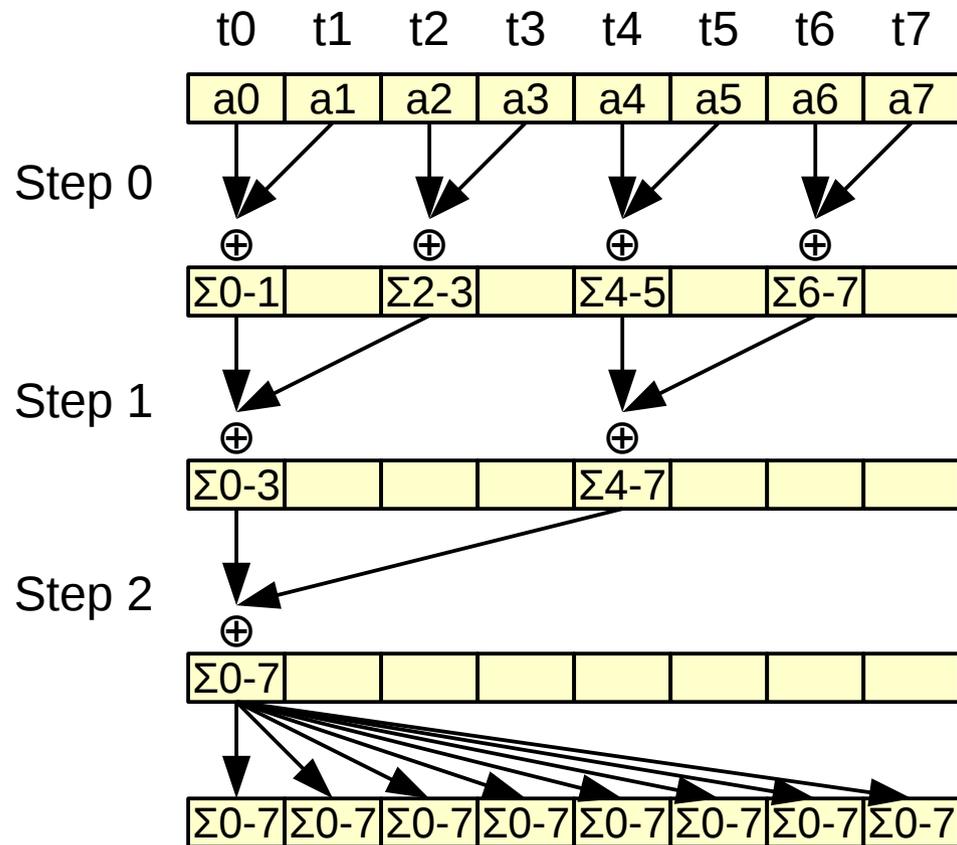
# Agenda

---

- 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



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

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

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

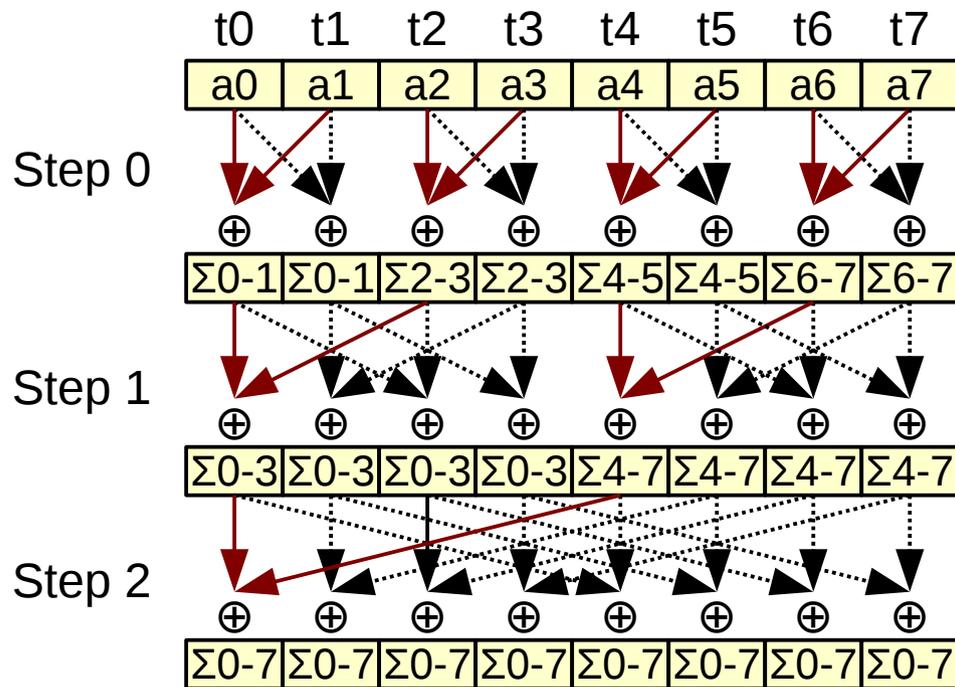
$$a[i] \leftarrow a[0]$$

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

- Let's rewrite it using shuffles

# Example 1: reduction + broadcast

Using butterfly shuffle



$a_i$  is a register:  
no memory access!

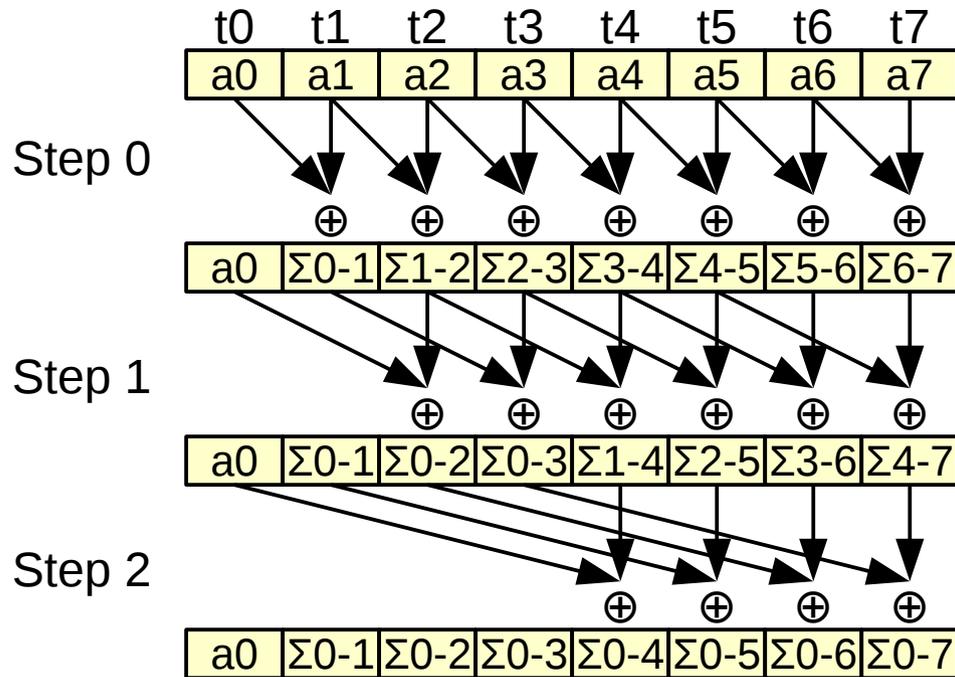
$a_i += g.shfl\_xor(a_i, 1);$

$a_i += g.shfl\_xor(a_i, 2);$

$a_i += g.shfl\_xor(a_i, 4);$

# Example 2: parallel prefix

## Kogge-Stone algorithm



```
s[i] ← a[i]
if i ≥ 1 then
    s[i] ← s[i-1] + s[i]
```

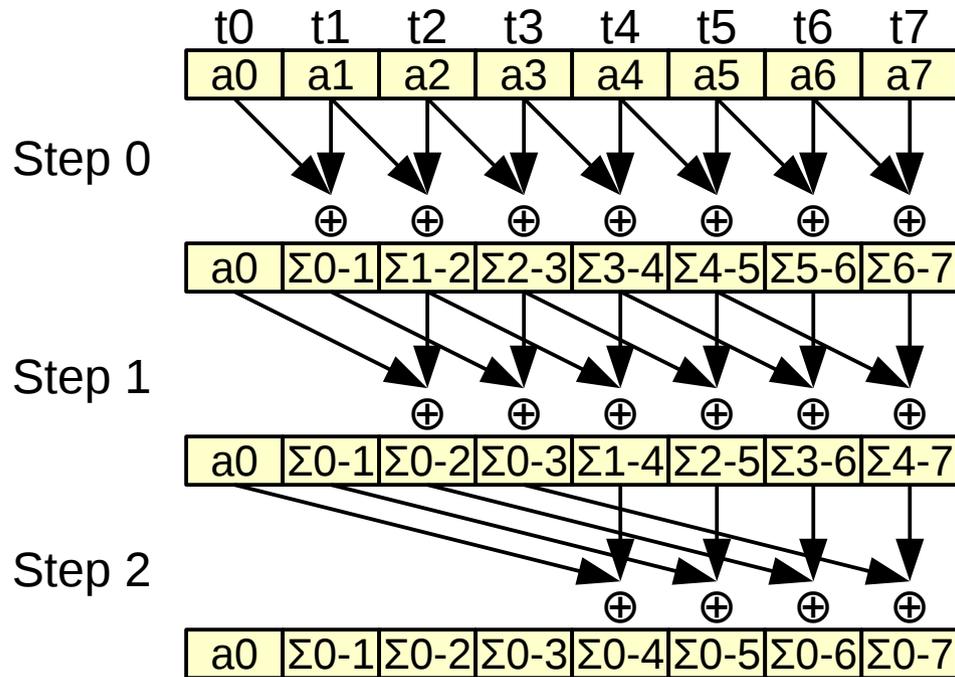
```
if i ≥ 2 then
    s[i] ← s[i-2] + s[i]
```

```
if i ≥ 4 then
    s[i] ← s[i-4] + s[i]
```

```
Step d: if i ≥ 2d then
        s[i] ← s[i-2d] + 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;
```

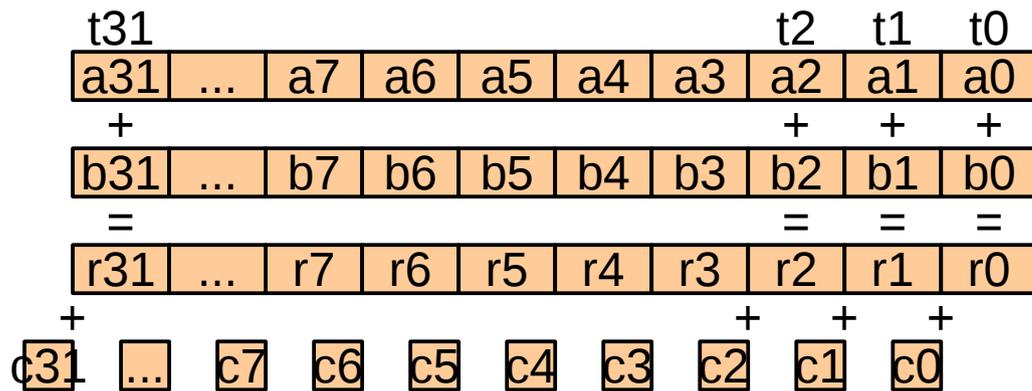
```
for(d = 1; d < 8; d *= 2) {  
    n = g.shfl_up(s, d);  
    if(g.thread_rank() >= d)  
        s += n;  
}
```

g.shfl\_up does implicit sync:  
must stay **outside** divergent if!

# Example 3: multi-precision addition

Add two 1024-bit integers together

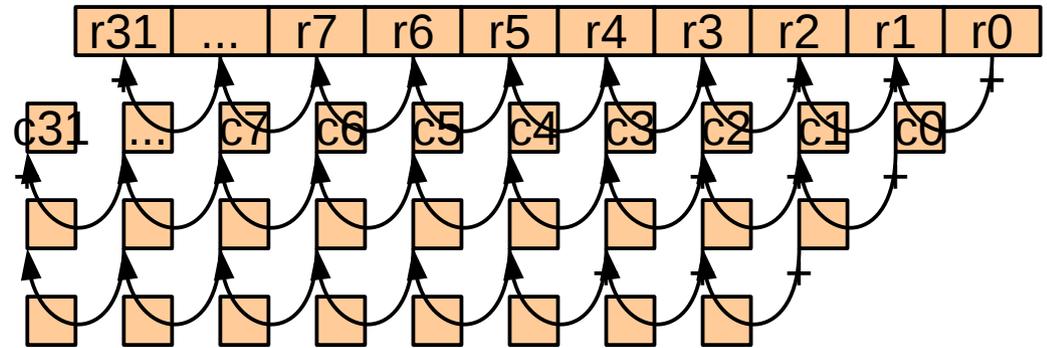
- Represent big integers as vectors of 32×32-bit
  - ◆ A warp works on a single big integer
  - ◆ Each thread works on a 32-bit digit
- First step: add vector elements in parallel and recover carries



```
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



```
uint32_t a = A[tid],  
         b = B[tid], r, c;  
  
r = a + b;    // Sum  
c = r < a;    // Get carry  
while(g.any(c)) { // Carry left?  
    c = g.shfl_up(c, 1); // Move left  
    if(g.thread_rank() == 0) c = 0;  
    r = r + c;    // Sum carry  
    c = r < c;    // New carry?  
}  
R[tid] = r;
```

# Bonus: propagating carries using +

- 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...

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

```
r = a + b;    // Sum  
c = 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 and add carry  
R[tid] = r;
```

e.g. in decimal:

a: 

2	0	6	2	3	8	7	1	9	4
---	---	---	---	---	---	---	---	---	---

b: 

7	0	6	1	6	1	4	3	0	5
---	---	---	---	---	---	---	---	---	---

r: 

9	0	2	3	9	9	1	4	9	9
---	---	---	---	---	---	---	---	---	---

c/gen: 

0	0	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---

prop/gen: 

1	0	1	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---	---	---

gen+(...): 

1	1	0	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---

(...)^prop: 

0	1	0	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---

→ 

9	1	2	4	0	0	1	4	9	9
---	---	---	---	---	---	---	---	---	---

# Agenda

---

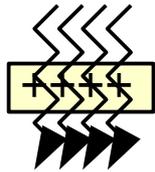
- Introducing cooperative groups
  - ◆ API overview
  - ◆ Thread block tile
- Collective operations
  - ◆ Shuffle
  - ◆ Vote
  - ◆ Match
- Thread block tile examples
  - ◆ Reduction
  - ◆ Parallel prefix
  - ◆ Multi-precision addition
- Coalesced groups
  - ◆ Motivation
  - ◆ ~~Example~~: stream compaction

# What about divergence management?

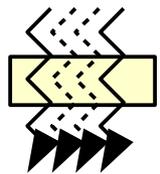
SIMT model (*NVIDIA GPUs*)



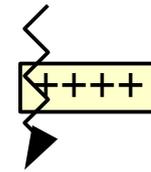
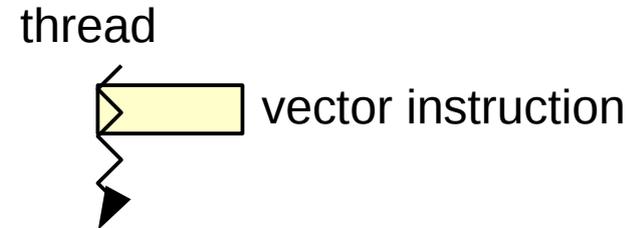
- Common denominator: independent calculations



- **Feature: automatic branch divergence management**



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



- **Feature: direct communication across SIMD lanes**



- Thread block tiles enable direct inter-lane communication
  - What about branch divergence?

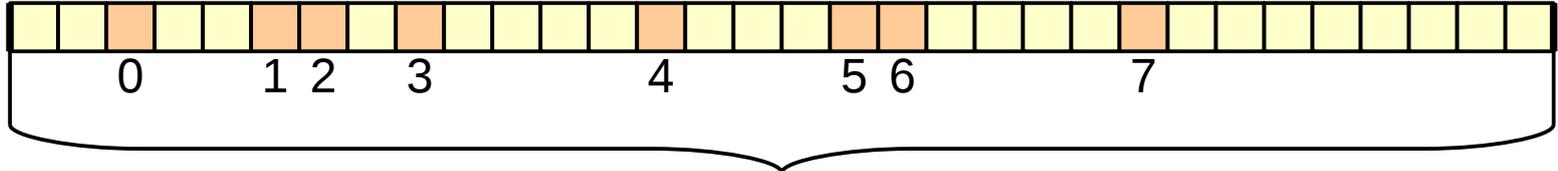
# Coalesced group

- 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

```
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

g.thread\_rank:



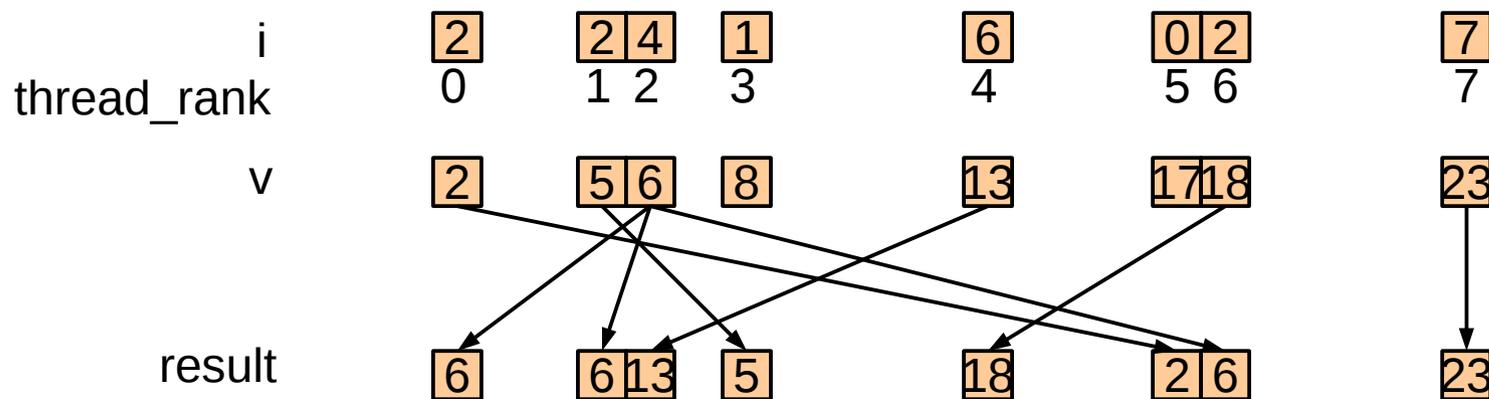
g.size() = 8

Warp



# Collective operations on coalesced groups

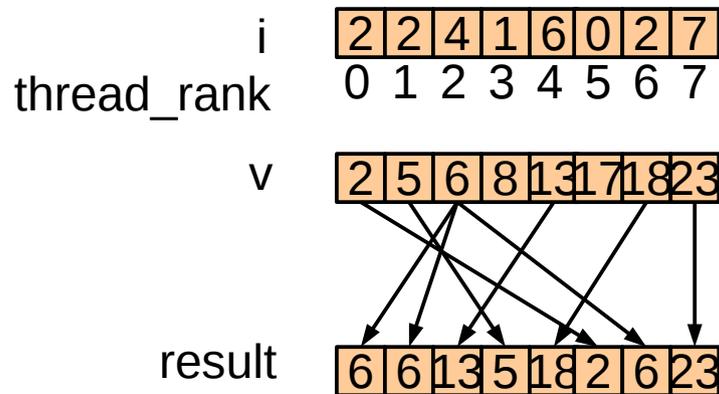
- Support full assortment of shuffle, vote, match!
  - ◆ All indexing is based on computed thread rank
  - ◆ e.g. `g.shfl(v, i)`:



- 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)`:

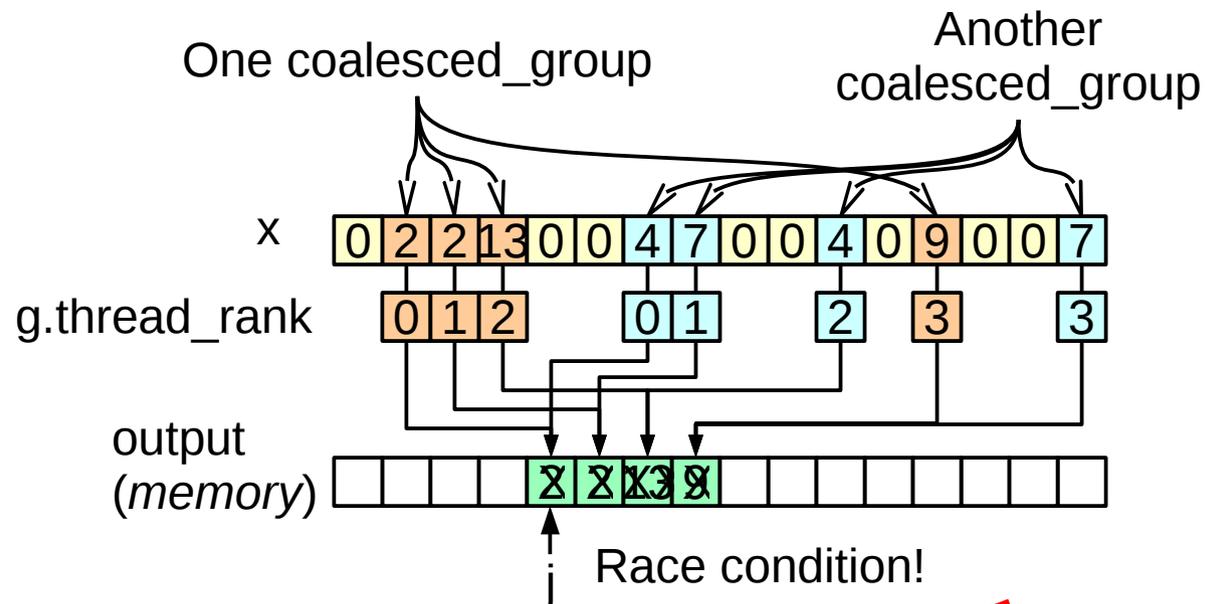


- You can just ignore inactive threads
- Beware of performance impact of thread rank remapping!



# 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!



```
__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;  
}
```

- **Unspecified** behavior: *it might even seem to work*

# The official `coalesced_group` example

Reference use case from CUDA documentation

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

```
__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;
}
```

Implicit sync



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

# 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

```
__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!

```
__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);`
- No unified inter-thread communication primitives across all groups yet
  - ➔ Thread block group primitives as an abstraction over shared memory?
- 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*.  $\forall$  blog  
<https://devblogs.nvidia.com/paralleforall/cooperative-groups/>
- Yuan Lin, Vinod Grover. *Using CUDA Warp-Level Primitives*  $\forall$  blog  
<https://devblogs.nvidia.com/using-cuda-warp-level-primitives/>