COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

Cooperative Groups: a flexible model for synchronization and communication within groups of threads.

At a glance

Scalable Cooperation among groups of threads

Flexible parallel decompositions

Composition across software boundaries

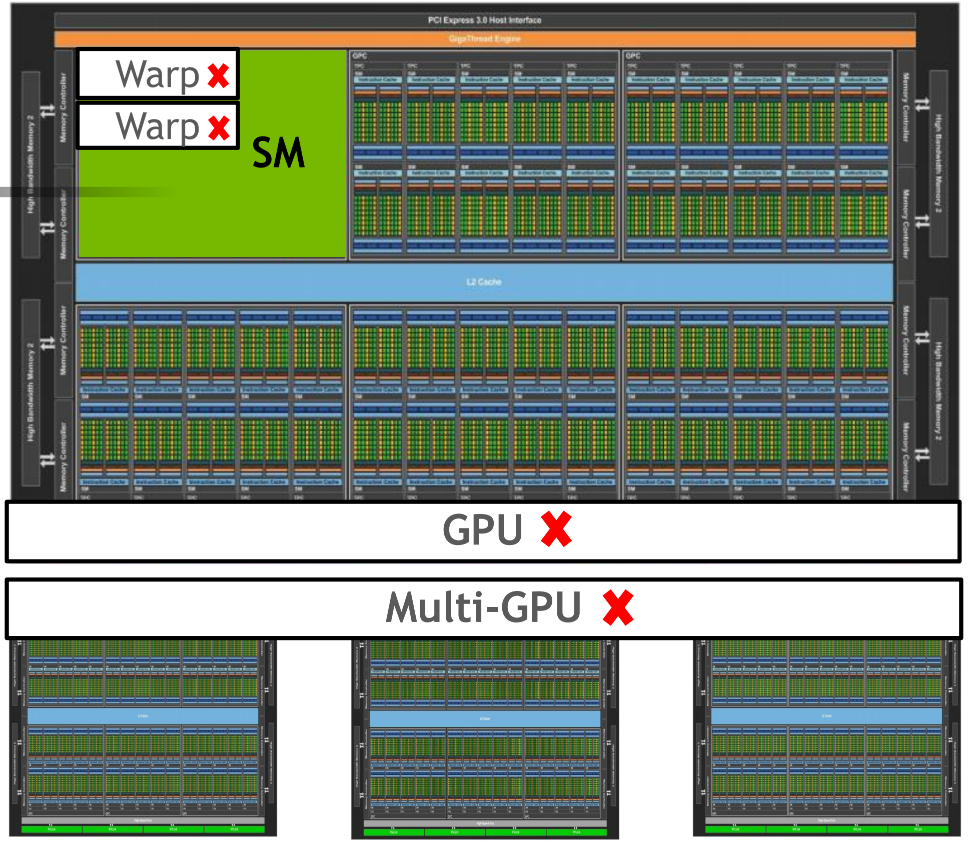
Deploy Everywhere

Benefits all applications

Examples include:
- Persistent RNNs
- Physics
- Search Algorithms
- Sorting

LEVELS OF COOPERATION: TODAY

`__syncthreads()`: block level synchronization barrier in CUDA



LEVELS OF COOPERATION: CUDA 9.0

For current coalesced set of threads:

```
  auto g = coalesced_threads();
```

For warp-sized group of threads:

```
 auto block = this_thread_block();
 auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

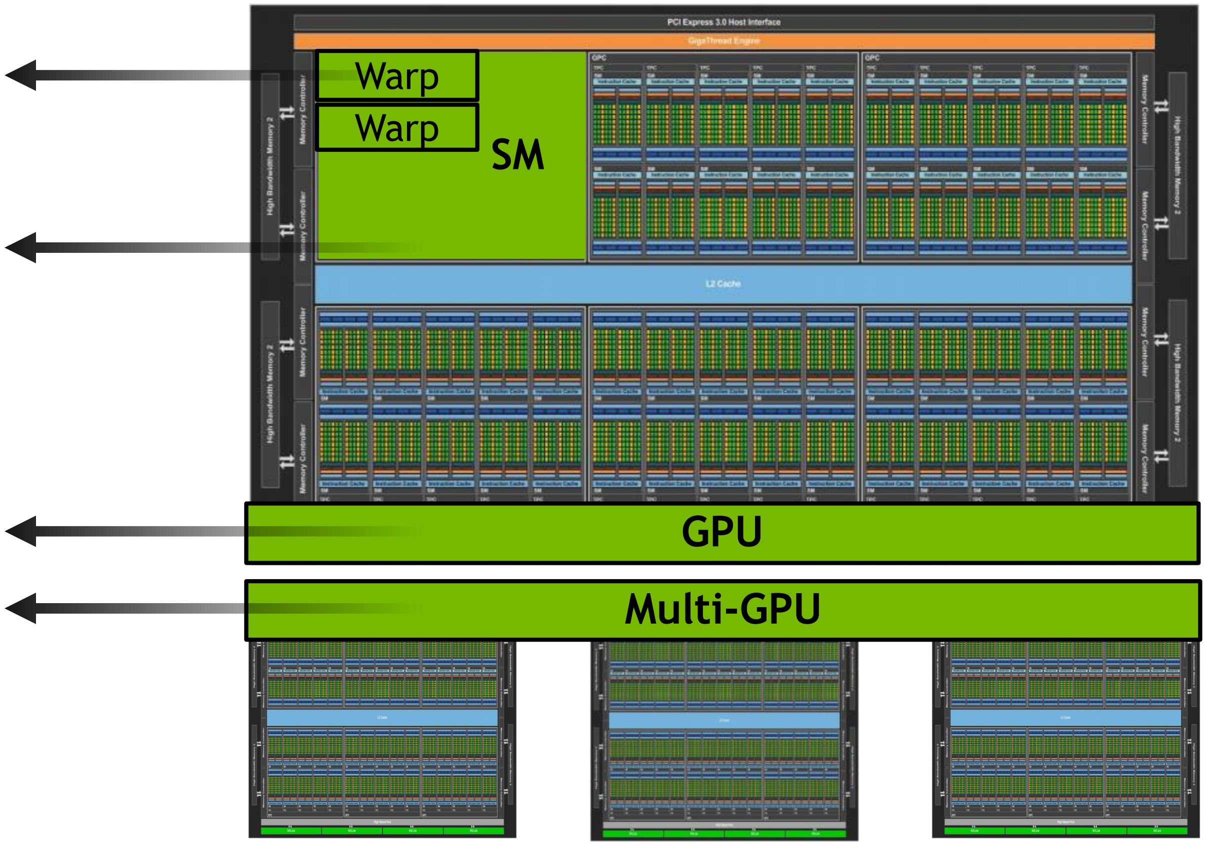
```
  auto g = this_thread_block();
```

For device-spanning grid:

```
  auto g = this_grid();
```

For multiple grids spanning GPUs:

```
  auto g = this_multi_grid();
```



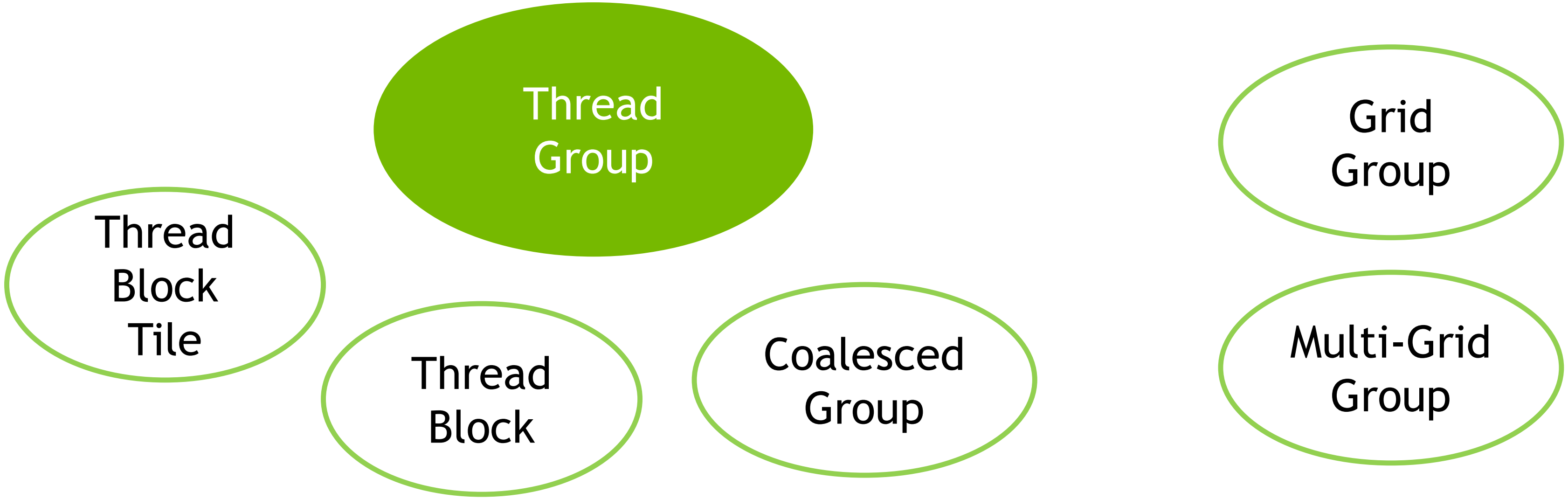
All Cooperative Groups functionality is within a cooperative_groups:: namespace

THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as thread_group:

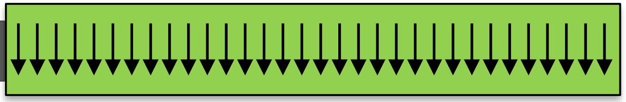
```
void sync();                    // Synchronize the threads in the group
unsigned size();                // Total number of threads in the group
unsigned thread_rank();         // Rank of the calling thread within [0, size]
bool is_valid();                // Whether the group violated any API constraints
```

And additional thread_block specific functions:

```
dim3 group_index();             // 3-dimensional block index within the grid
dim3 thread_index();            // 3-dimensional thread index within the block
```

PROGRAM DEFINED DECOMPOSITION

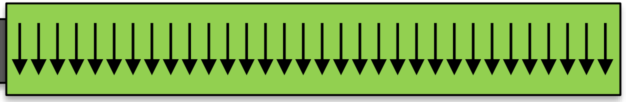
CUDA KERNEL



All threads launched

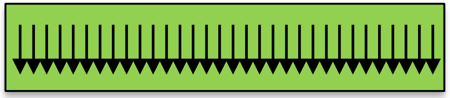
```
thread_block g = this_thread_block();
```

foobar(thread_block g)

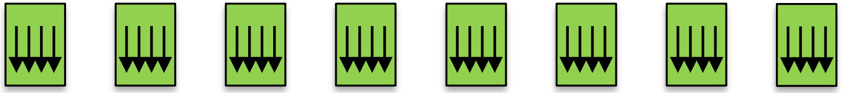


All threads in thread block

```
thread_group tile32 = tiled_partition(g, 32);
```



```
thread_group tile4 = tiled_partition(tile32, 4);
```



Restricted to powers of two, and <= 32 in initial release

GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;       g.sync();
    val += x[lane + i];  g.sync();
  }
  return val;
}
```

THREAD BLOCK TILE

A subset of threads of a thread block, divided into tiles in row-major order

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());
```

```
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

Exposes additional functionality:

```
.shfl()
.shfl_down()
.shfl_up()
.shfl_xor()
```

```
.any()
.all()
.ballot()
.match_any()
.match_all()
```

Size known at compile time = fast!

STATIC TILE REDUCE

Per-Tile of 16 threads

```
g = tiled_partition<16>(this_thread_block());
tile_reduce(g, myVal);
```

```
template <unsigned size>
__device__ int tile_reduce(thread_block_tile<size> g, int val) {
  for (int i = g.size()/2; i > 0; i /= 2) {
    val += g.shfl_down(val, i);
  }
  return val;
}
```

GRID GROUP

A set of threads within the same grid, guaranteed to be resident on the device

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernel(…)`

```
__global__ kernel() {
    grid_group grid = this_grid();
    // load data
    // loop - compute, share data
        grid.sync();
        // devices are now synced
}
```

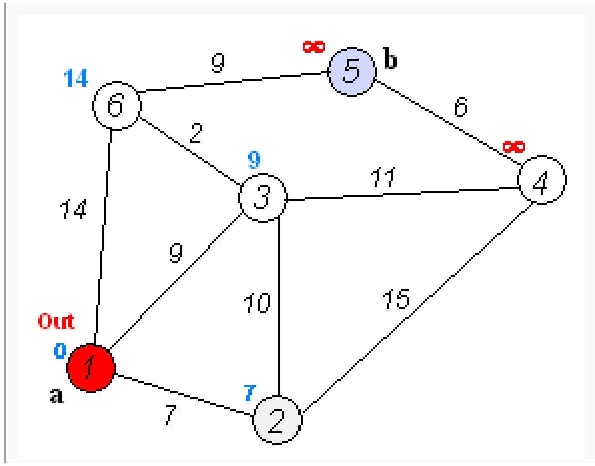
Device needs to support the `cooperativeLaunch` property.

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, kernel, numThreads, 0));
```

GRID GROUP

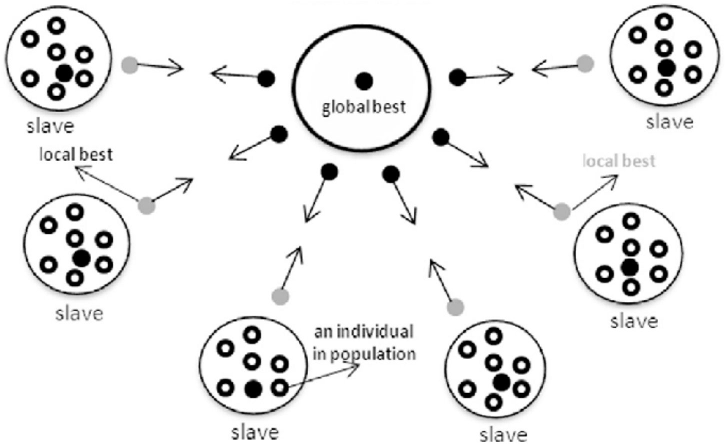
The goal: keep as much state as possible resident

Shortest Path / Search



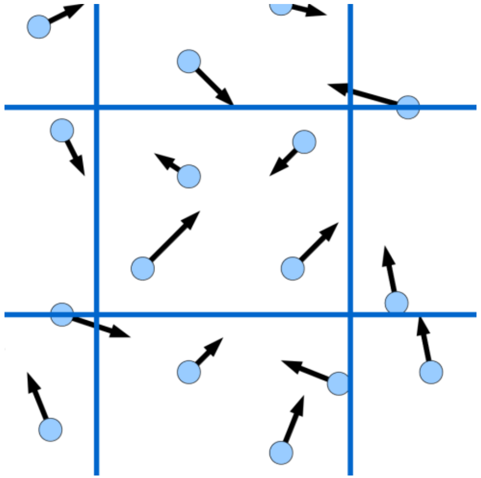
Weight array perfect for persistence
Iteration over vertices? Fuse!

Genetic Algorithms / Master driven algorithms



Synchronization between a master block and slaves

Particle Simulations

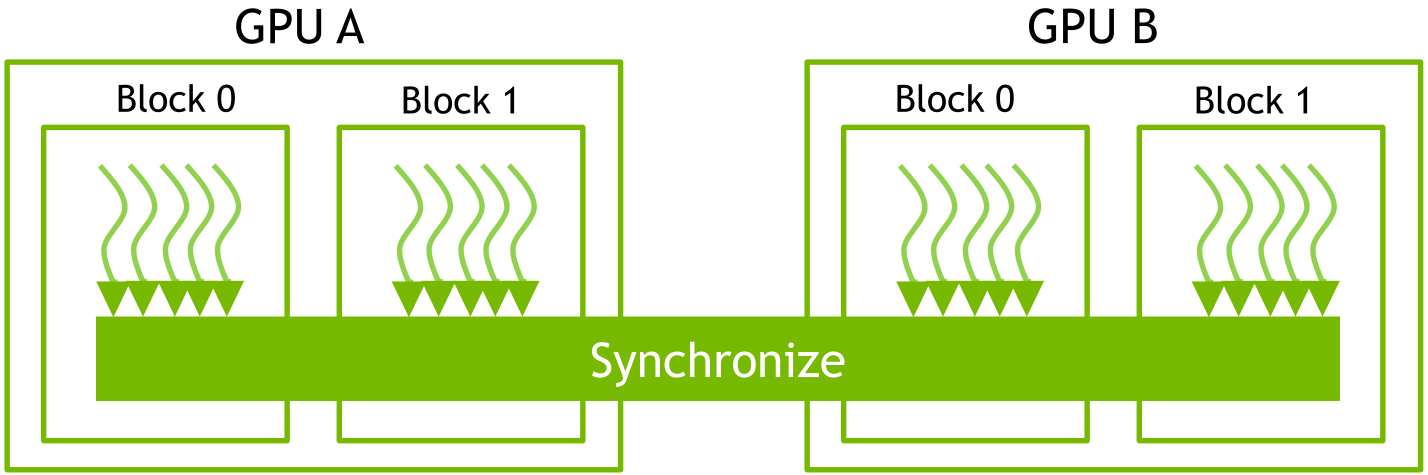


Synchronization between update and collision simulation

MULTI GRID GROUP

A set of threads guaranteed to be resident on the same system, on multiple devices

```
__global__ void kernel() {
    multi_grid_group multi_grid = this_multi_grid();
    // load data
    // loop - compute, share data
        multi_grid.sync();
        // devices are now synced, keep on computing
}
```



MULTI GRID GROUP

Launch on multiple devices at once

New CUDA Launch API to opt-in:

`cudaLaunchCooperativeKernelMultiDevice(…)`

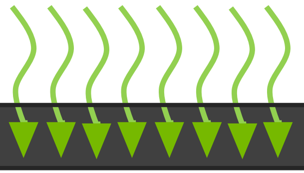
Devices need to support the `cooperativeMultiDeviceLaunch` property.

```
struct cudaLaunchParams params[numDevices];
for (int i = 0; i < numDevices; i++) {
    params[i].func = (void *)kernel;
    params[i].gridDim = dim3(…); // Use occupancy calculator
    params[i].blockDim = dim3(…);
    params[i].sharedMem = …;
    params[i].stream = …; // Cannot use the NULL stream
    params[i].args = …;
}
cudaLaunchCooperativeKernelMultiDevice(params, numDevices);
```

COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

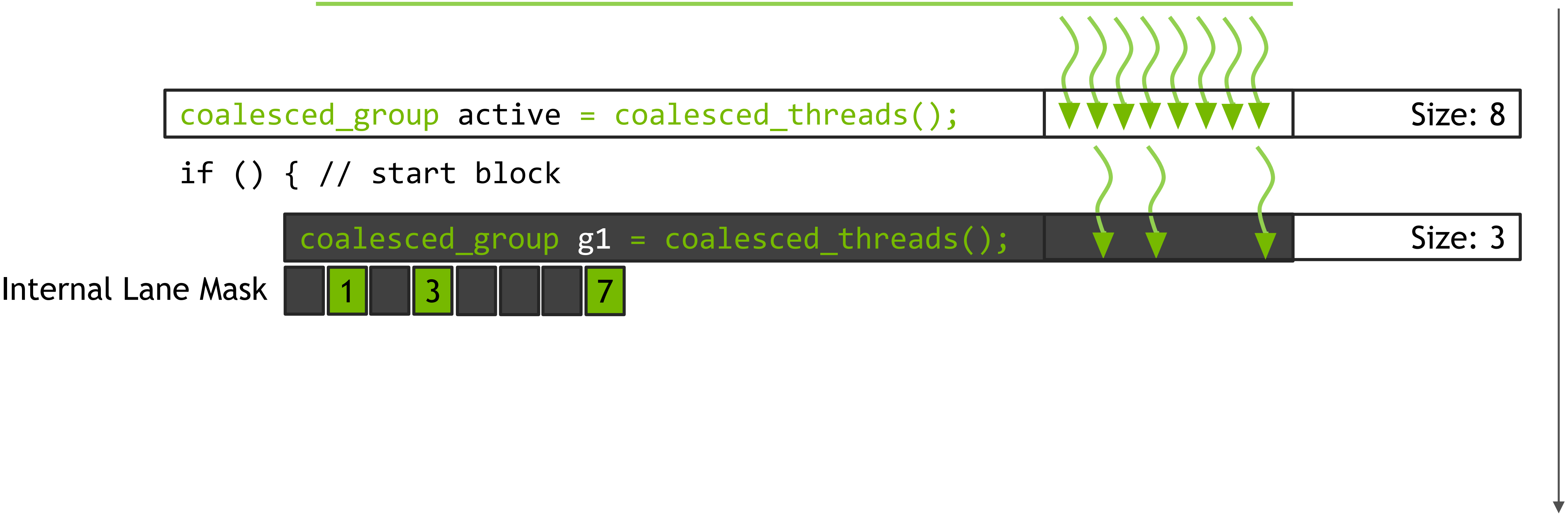
```
coalesced_group active = coalesced_threads();
```



Size: 8

COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

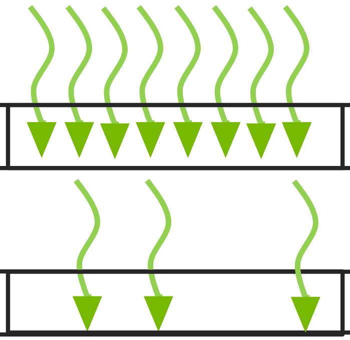
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

```
coalesced_group active = coalesced_threads();
```

Size: 8

```
if () { // start block
    coalesced_group g1 = coalesced_threads();
```

Size: 3



Internal Lane Mask

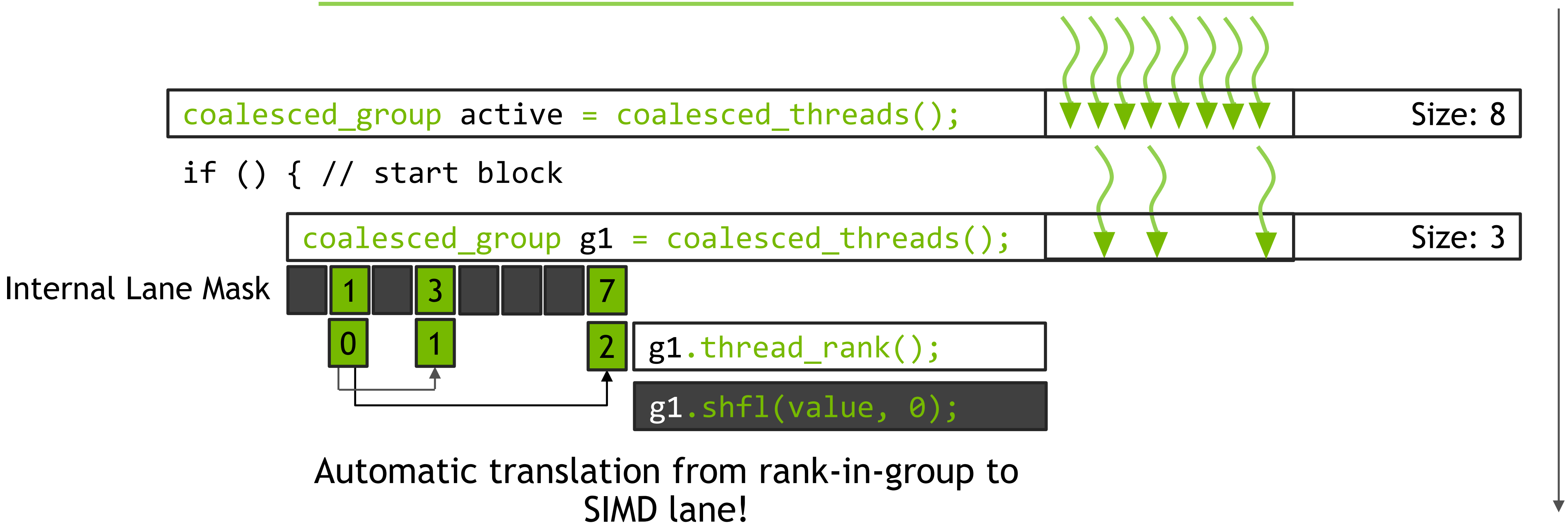
	1		3				7
	0		1				2

```
g1.thread_rank();
```

Automatic translation to rank-in-group!

COALESCED GROUP

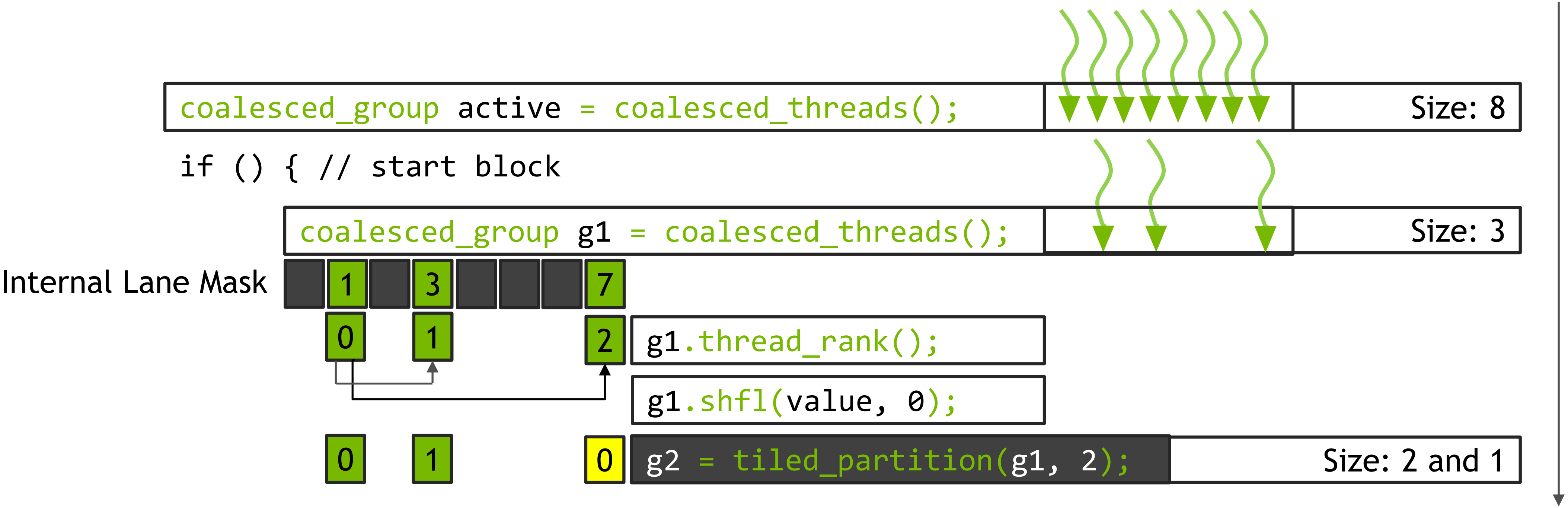
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation from rank-in-group to SIMD lane!

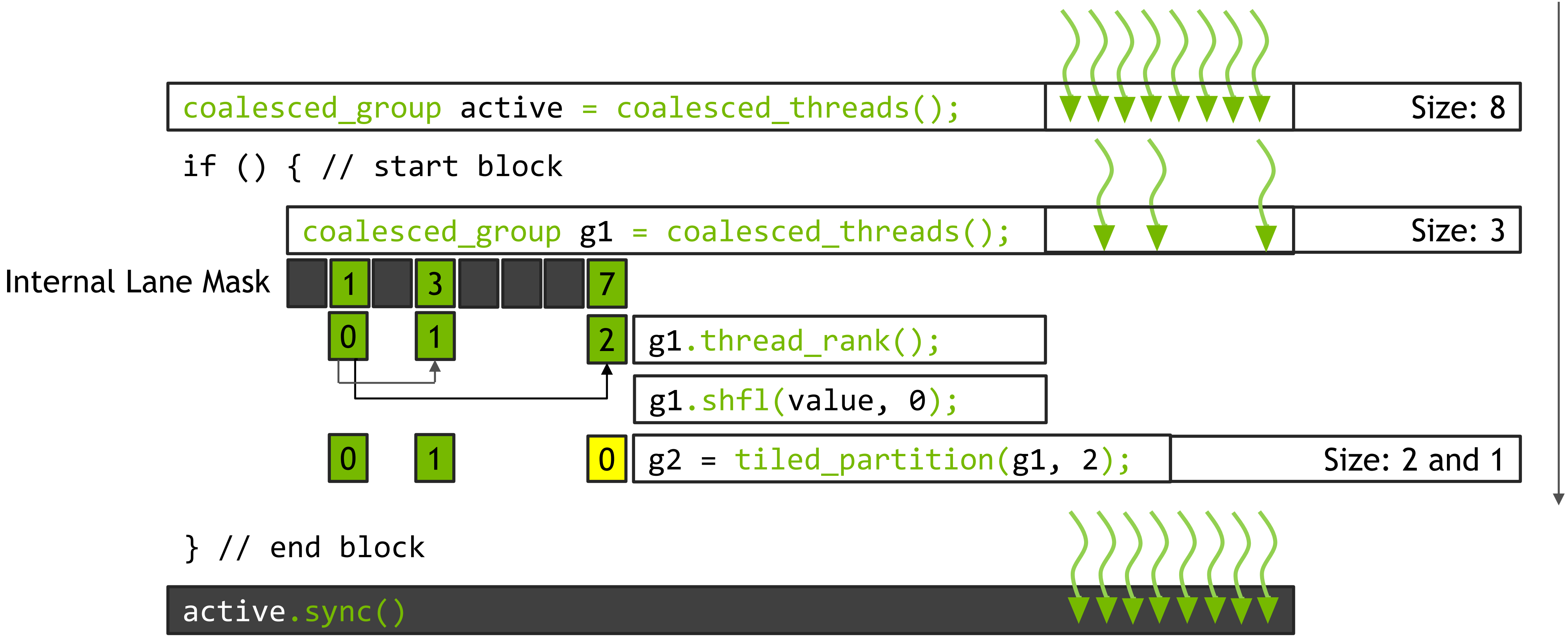
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

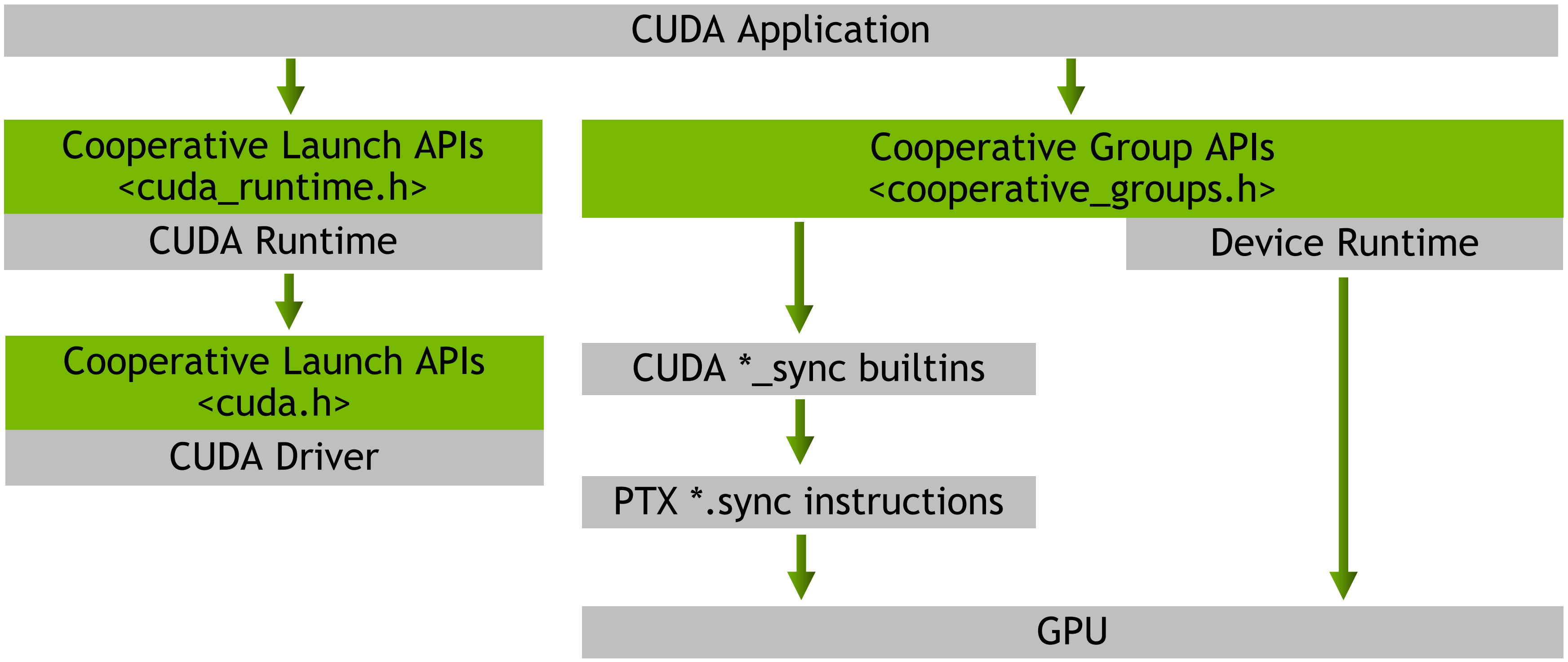


ATOMIC AGGREGATION

Opportunistic cooperation within a warp

```
inline __device__ int atomicAggInc(int *p)
{
    coalesced_group g = coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.size());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

ARCHITECTURE



WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

cuda warp synchronous programming

All | Videos | Images | Shopping | News | More | Settings | Tools

About 17,200 results (0.27 seconds)

[PDF] Warp-synchronous programming - Irisa
www.irisa.fr/alf/downloads/collange/cours/gpuprog_ufmg.../gpu_ufmg_2015_5.pdf
Mar 9, 2016 - Lane ID exists in PTX, not in C for **CUDA**. Can be recovered using ..... If you know **warp-synchronous programming** in **CUDA**, you know SIMD ...

Warp synchronous programming - NVIDIA Developer Forums
https://devtalk.nvidia.com/default/topic/807699/warp-synchronous-programming/
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/19011826/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 ...

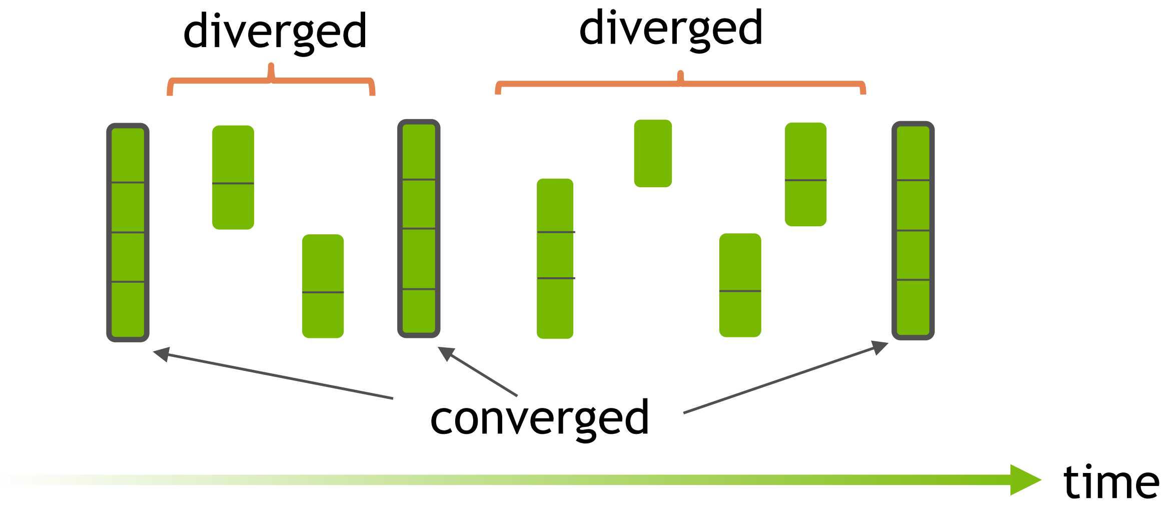
parallel processing - CUDA __syncthreads() usage within a warp ...
stackoverflow.com/questions/10205245/cuda-syncthreads-usage-within-a-warp
Apr 18, 2012 - Updated with more information about using volatile. Presumably you want all threads ... On the face of it this means you can omit the __syncthreads() , 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.

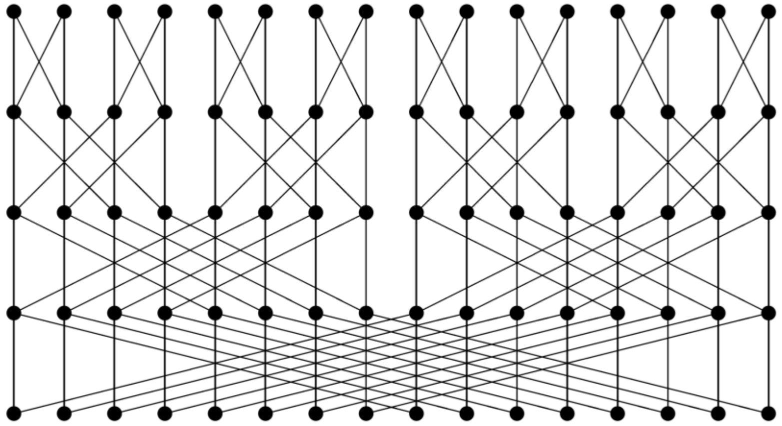
- e.g. reduction, scan, aggregated atomic operation, etc.

CUDA C++ supports warp synchronous programming by providing warp synchronous built-in functions and cooperative group collectives.

EXAMPLE: SUM ACROSS A WARP

```
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);
```

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

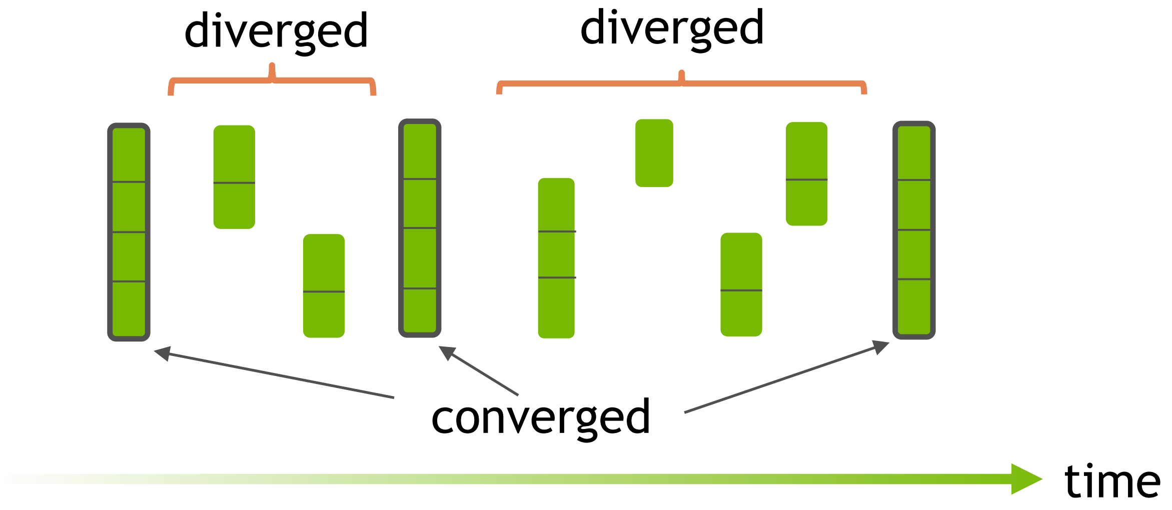


HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

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

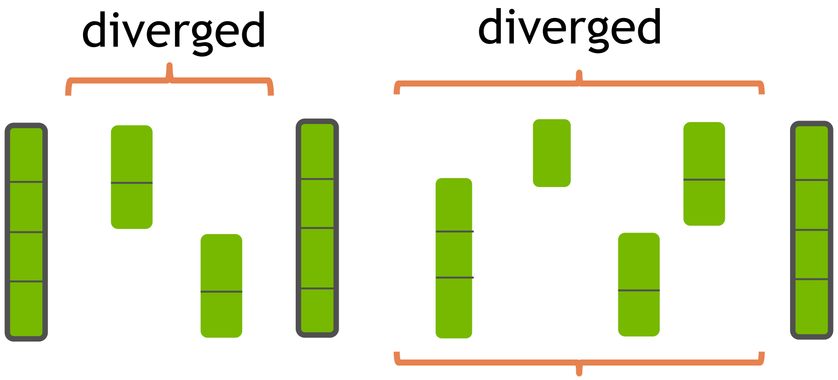
Make Sync Explicit

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

- __syncwarp

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

Find the active threads

```
// 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 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

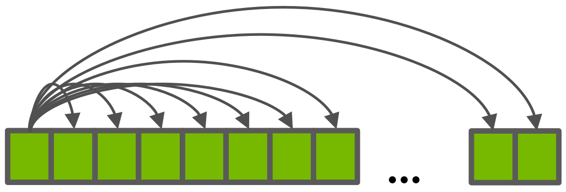
Returns true when all threads in ‘mask’ have the same predicate value

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);
```

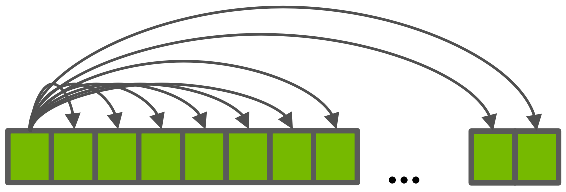


EXAMPLE: SHUFFLE

__shfl_sync, __shfl_down_sync

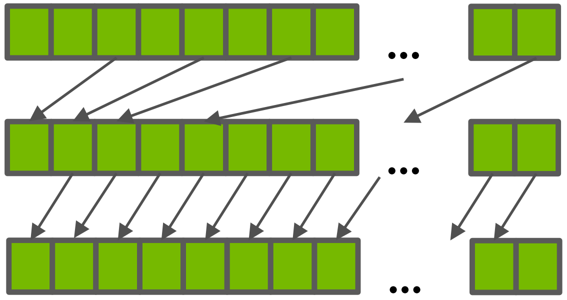
Broadcast: all threads get the value of ‘x’ from lane id 0

```
y = __shfl_sync(0xffffffff, x,  0);
```



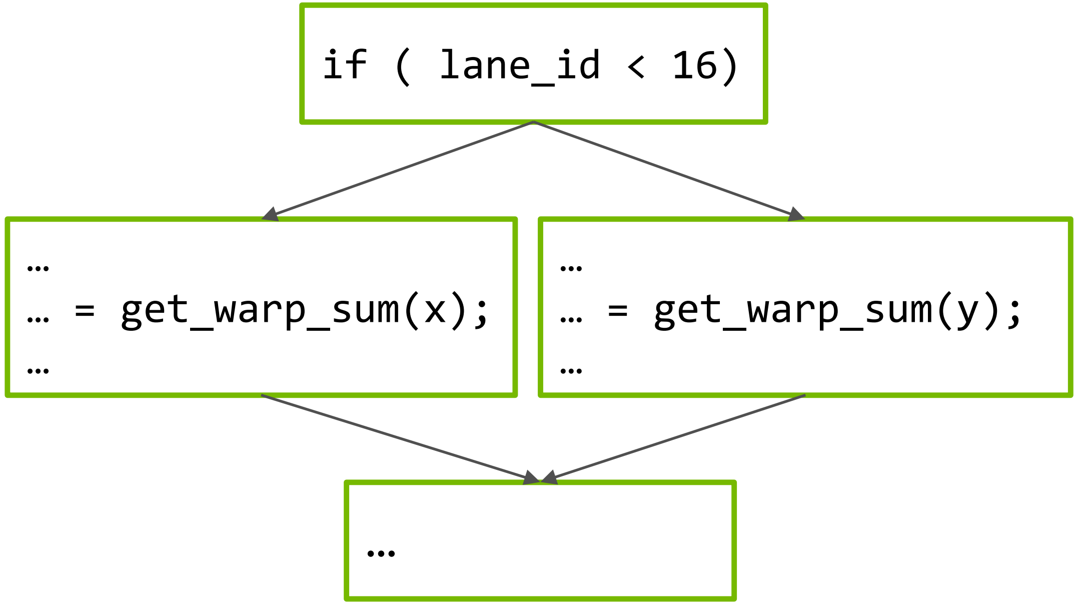
Reduction:

```
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

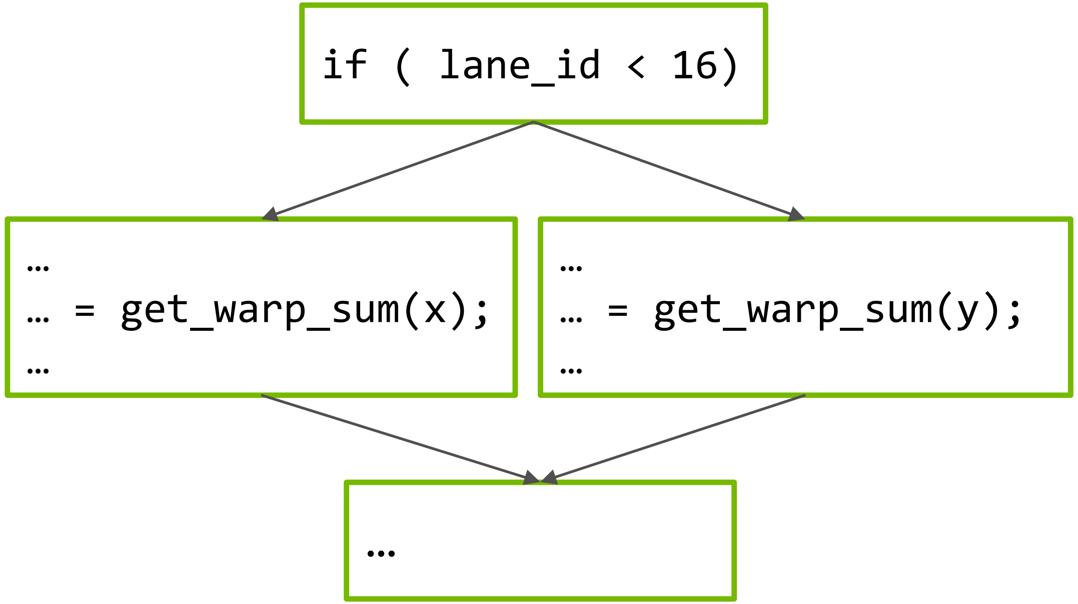



```
#define FULLMASK 0xffffffff

__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




```
#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 += 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;
```

BUT WHAT'S WRONG WITH THIS CODE?

```
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

Unsafe and Unsupported

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:

```
if (lane_id < 16)
   A;
else
   B;
assert(__activemask() == 0xffffffff);
```

IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```
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

```
if (__activemask() == 0xffffffff) {
    assert(__activemask() == 0xffffffff);
}
```

IMPLICIT 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

```
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];
```

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

- Make sync explicit

```
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;
```

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

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

```
coalesced_group g = coalesced_threads();



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

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

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

All threads in thread block must arrive at this barrier.

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

Hidden constraint on caller due to implementation of *sum*.

BETTER COMPOSITION

Explicit cooperative interfaces

```
__device__ int sum(thread_group g, int *x, int n)
{
    ...
    g.sync()
    ...
    return total;
}

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

Participating thread group provided by caller.

The need to synchronize in *sum* is visible in code.

FUTURE ROADMAP

Partition by label or predicate, more complex scopes

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

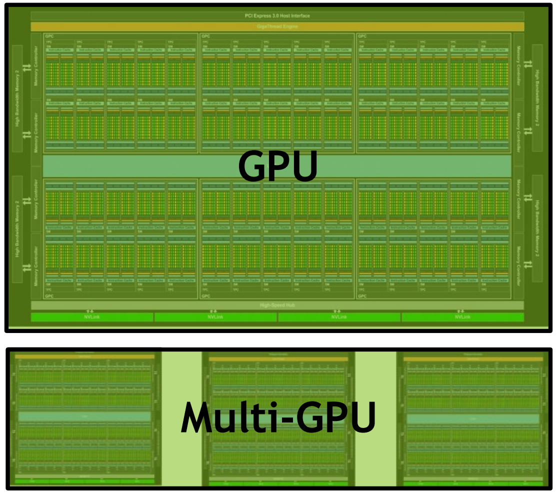
```
thread_group cta = this_thread_block();
thread_group g = partition(cta, cta.thread_rank() & 1);
```

Warp 32 | Warp 32 | Warp 32

0 0 0 0 … | 0 0 0 0 … | 1 1 1 1 …

```
thread_group g = tiled_partition(cta, 64);
```

At all scopes!



FUTURE ROADMAP

Library of collectives (sort, reduce, etc.)

```
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> group_reduce;

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

On a simpler note:

```
// 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