COOPERATIVE GROUPS
FLEXIBLE GROUPS OF THREADS

30 April 2021 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Overview, Outline

At a Glance

- Cooperative Groups: New model to work with thread groups
- Thread groups are entities, intrinsic function as member functions
- Safe and structured programming

Contents

Gather Last-Minute Material

Now run

```
jsc-material-reset-08
jsc-material-reset-09
jsc-material-reset-10
jsc-material-reset-11
```

Gather Last-Minute Material

Now run

```
jsc-material-reset-08
jsc-material-reset-09
jsc-material-reset-10
jsc-material-reset-11
```

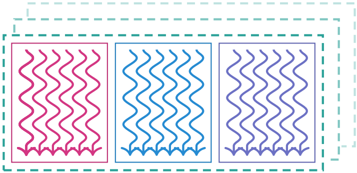
Place cursor in box when done:

I’m done!

Standard CUDA Threading Model

Before CUDA 9

- Many threads, combined into blocks, on a grid; in 3D
- Operation: Single Instruction, Multiple Threads (SIMT)
- Thread waiting for result of instruction? Use computational resource with other threads in meantime!
- Group of threads execute in lockstep: **Warp** (currently 32 threads)
  - Same instructions
  - Branching possible
  - Predicates (and masks)
- Shared memory: Fast, shared between threads of block
- Synchronization between threads of blocks:
  `__syncthreads()` – barrier for all threads of block



Cooperative Groups

Introduction

New Model: Cooperative Groups

- Motivation to extend classical model

Algorithmic	Not all algorithms map easily to available synchronization methods; **synchronization** should be more flexible
Design	Make groups of threads explicit **entities**
Hardware	Access new **hardware features** (*Independent Thread Scheduling*)

→ **Cooperative Groups** (CG)

A flexible model for synchronization and communication within groups of threads.

New Model: Cooperative Groups

- Motivation to extend classical model

Algorithmic	Not all algorithms map easily to available synchronization methods; **synchronization** should be more flexible
Design	Make groups of threads explicit **entities**
Hardware	Access new **hardware features** (*Independent Thread Scheduling*)

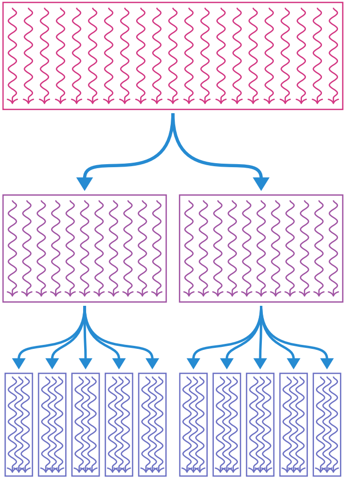
→ **Cooperative Groups** (CG)

A flexible model for synchronization and communication within groups of threads.

- All in `namespace cooperative_groups` (`cooperative_groups.h` header)
- Following in text: `cooperative_groups::func()` ⟶ `cg::func()`

```
namespace cg = cooperative_groups;
```

Division of Thread Blocks



- Start with block of certain size
- Divide into smaller sub-groups
- Continue diving, if algorithm makes it necessity
- Methods for dynamic or static divisions (*tiles*)
- In each level: thread of group has unique ID (local index instead of global index)

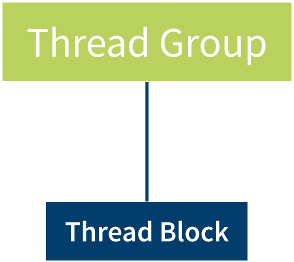
Cooperative Groups

Thread Groups Overview

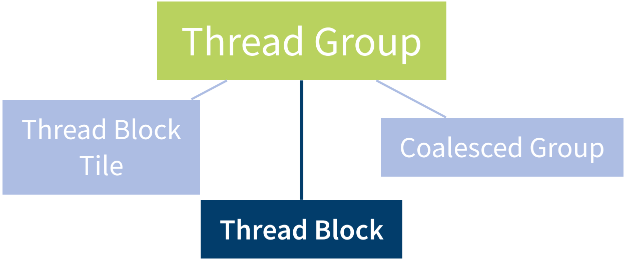
Thread Group Landscape

Thread Group

Thread Group Landscape

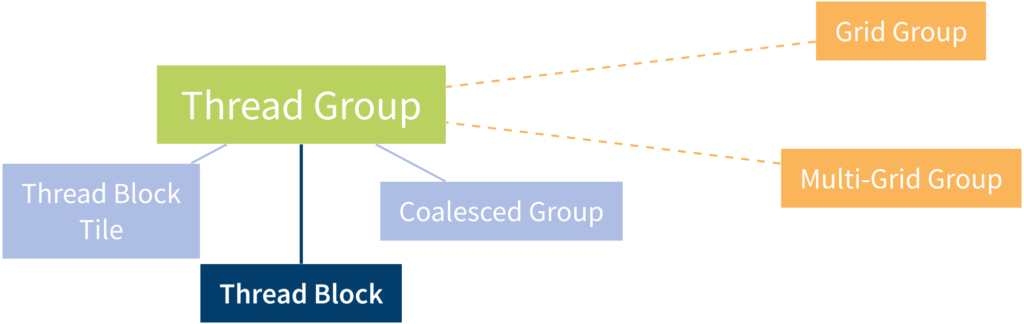


Thread Group Landscape



JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Thread Group Landscape



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Common Methods of Cooperative Groups

- Fundamental type: `thread_group`
- Every CG has following member functions

`sync()` Synchronize the threads of this group (alternative `cg::sync(g)`)
Before: `__syncthreads()` for whole block

`thread_rank()` Get unique ID of current thread in this group (*local index*)
Before: `threadIdx.x` for index in block

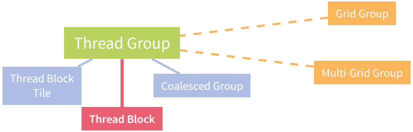
`size()` Number of threads in this group
Before: `blockDim.x` for number of threads in block

`is_valid()` *Group is technically ok*

Cooperative Groups

Thread Blocks

Cooperative Thread Blocks



- Easiest entry point to thread groups: `cg::this_thread_block()`
- Additional member functions

`thread_index()`	Thread index within block (3D)
`group_index()`	Block index within grid (3D)

- Blocks (and groups) are now concrete entities

→ Design functions to represent this!

Example: Print Rank Function

```
__device__ void printRank(cg::thread_group g) {
    printf("Rank %d\n", g.thread_rank());
}
__global__ void allPrint() {
    cg::thread_block b = cg::this_thread_block();

    printRank(b);
}
int main() {
    allPrint<<<1, 23>>>();
}
```

Task Base Code: Shared Memory Reduction

Outer skeleton

```
int * array;
cudaMallocManaged(&array, sizeof(int) * N);

for (int i = 0; i < N; i++)
    array[i] = rand() % 1024;

int blocks = 1;
int threads = N;
maxKernel<<<blocks, threads, threads * sizeof(int)>>>(array);
```

Allocate this much shared memory per block

Task Base Code: Shared Memory Reduction

Inner logic: Kernel

```
__global__ void maxKernel(int * array) {
    extern __shared__ int shmem_temp[];  // threads * sizeof(int)

    int threadIndex = threadIdx.x;
    int myValue = array[threadIndex];

    int maxValue = maxFunction(shmem_temp, myValue);

    __syncthreads();
    if (threadIndex == 0)
        array[0] = maxValue;
}
```

Task Base Code: Shared Memory Reduction

Inner logic: Kernel

```
__global__ void maxKernel(int * array) {
    extern __shared__ int shmem_temp[];  // threads * sizeof(int)

    int threadIndex = threadIdx.x;
    int myValue = array[threadIndex];

    int maxValue = maxFunction(shmem_temp, myValue);

    __syncthreads();
    if (threadIndex == 0)
        array[0] = maxValue;
}
```

One value for each thread

Task Base Code: Shared Memory Reduction

Inner logic: Kernel

```
__global__ void maxKernel(int * array) {
    extern __shared__ int shmem_temp[];  // threads * sizeof(int)

    int threadIndex = threadIdx.x;
    int myValue = array[threadIndex];

    int maxValue = maxFunction(shmem_temp, myValue);

    __syncthreads();
    if (threadIndex == 0)
        array[0] = maxValue;
}
```

One value for each thread

Call function with temp array and thread-local value

Task Base Code: Shared Memory Reduction

Inner logic: Kernel

```
__global__ void maxKernel(int * array) {
    extern __shared__ int shmem_temp[];  // threads * sizeof(int)

    int threadIndex = threadIdx.x;
    int myValue = array[threadIndex];

    int maxValue = maxFunction(shmem_temp, myValue);

    __syncthreads();
    if (threadIndex == 0)
        array[0] = maxValue;
}
```

One value for each thread

Call function with temp array and thread-local value

Save max to array in global memory

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Task Base Code: Shared Memory Reduction

Inner logic: Function

```
__device__ int maxFunction(int * workspace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

Task Base Code: Shared Memory Reduction

Inner logic: Function

```
__device__ int maxFunction(int * workspace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

Per loop, halve size of operations

Task Base Code: Shared Memory Reduction

Inner logic: Function

```
__device__ int maxFunction(int * workspace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

Per loop, halve size of operations

Get max from current thread and offset thread

Task Base Code: Shared Memory Reduction

Inner logic: Function

```
__device__ int maxFunction(int * workspace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

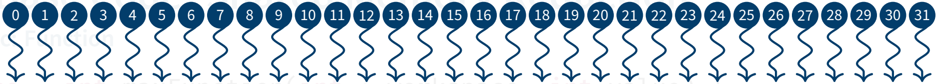
Put max value to current lane

Per loop, halve size of operations

Get max from current thread and offset thread

Task Base Code: Shared Memory Reduction

Inner logic: Function




```
__device__ int maxFunction(int * workspace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

Put max value to current lane

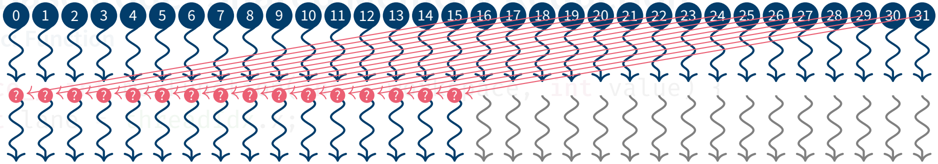
Per loop, halve size of operations

Get max from current thread and offset thread

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Task Base Code: Shared Memory Reduction

Inner logic: Function




```
__devic… …ace, int value) {
    int lane = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

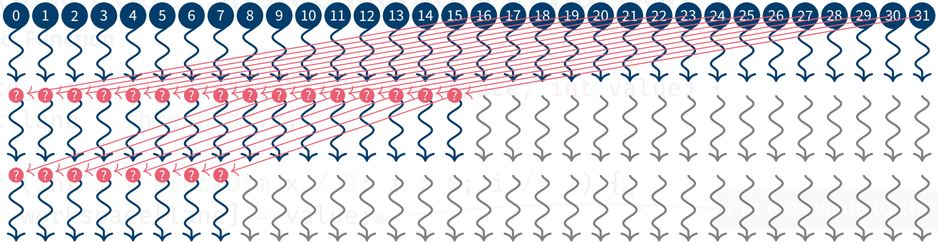
Put max value to current lane

Per loop, halve size of operations

Get max from current thread and offset thread

Task Base Code: Shared Memory Reduction

Inner logic: Function




```
    for (int i = blockDim.x / 2; i > 0; i /= 2) {
        workspace[lane] = value;

        __syncthreads();

        if (lane < i)
            value = max(value, workspace[lane + i]);

        __syncthreads();
    }
    return value;
}
```

Put max value to current lane

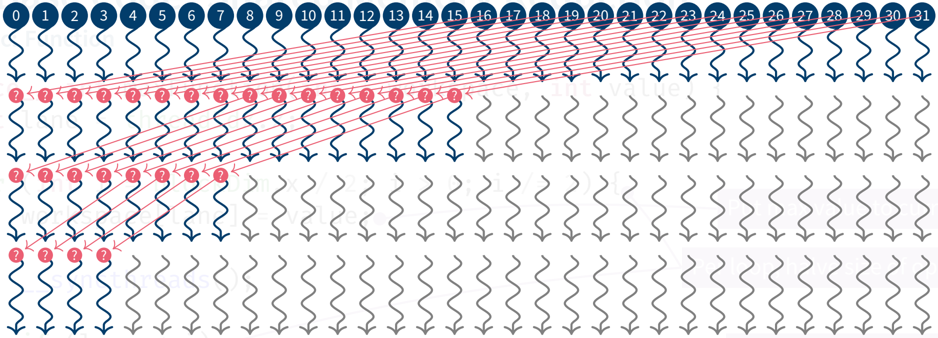
Per loop, halve size of operations

Get max from current thread and offset thread

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Task Base Code: Shared Memory Reduction

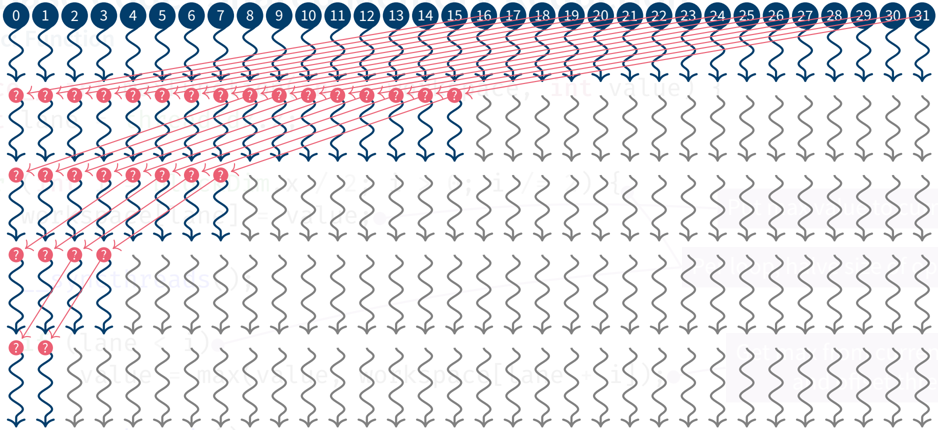
Inner logic Function



JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

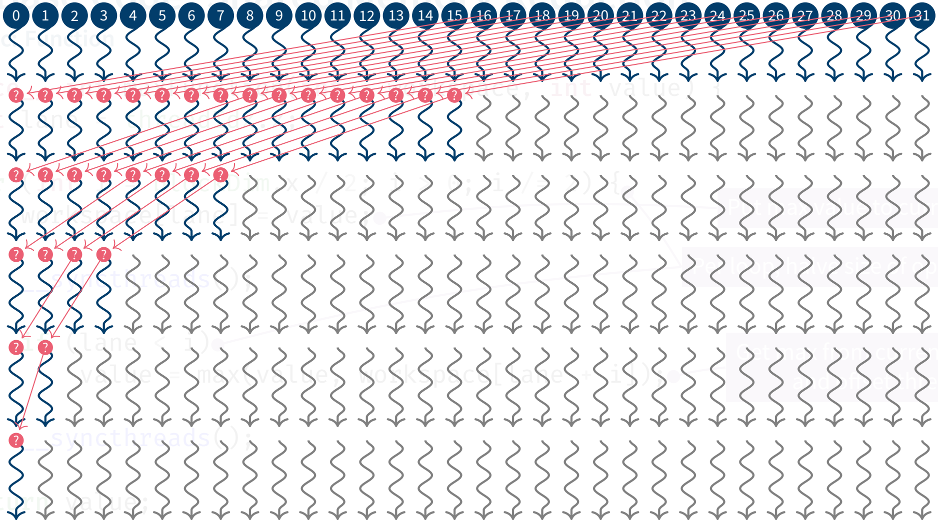
Task Base Code: Shared Memory Reduction

Inner logic Function



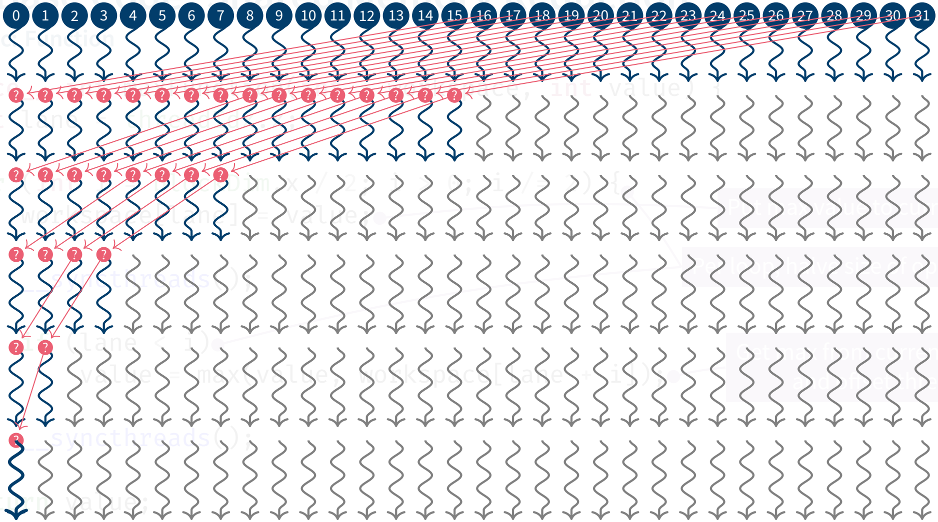
Task Base Code: Shared Memory Reduction

Inner logic: Function



Task Base Code: Shared Memory Reduction

Inner logic Function



Implementing a Cooperative Groups Kernel

TASK 1

From old to new

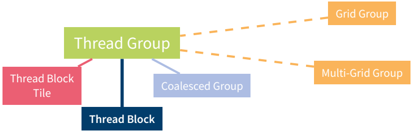
- Location of code: `8-Cooperative_Groups/exercises/tasks/task1`
- See `Instructions.md` for explanations
- Follow TODOs to port kernel/device function from traditional CUDA threading model to new CG model
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on Cooperative Groups

Cooperative Groups

Tiling Groups

Tiles of Groups

Dynamically-tiled



- Divide into smaller groups with `cg::tiled_partition()`
- Will automatically create smaller groups from parent group
- Examples
  - Create groups of size 32 of current block

```
cg::thread_group tile32 = cg::tiled_partition(cg::this_thread_block(), 32);
```

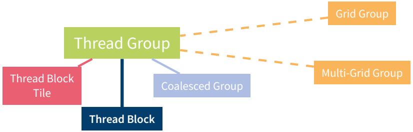
  - Create sub-groups of size 4

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

- **Note**: Currently, only supported partition sizes are 2, 4, 8, 16, 32

Tiles of Groups

Statically-tiled: `thread_block_tile`



- Second version of function: `cg::tiled_partition<>()`
- Size of tile is template parameter

→ Known at compile time! Optimizations possible!

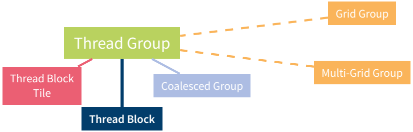
- Returns `thread_block_tile` object with additional member functions
  - `.shfl()`, `.shfl_down()`, `.shfl_up()`, `.shfl_xor()`
  - `.any()`, `.all()`, `.ballot()`; `.match_any()`, `.match_all()`

  → Intrinsic functions to work with threads inside a warp *(more later)*

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Tiles of Groups

Statically-tiled: `thread_block_tile`



- Second version of function: `cg::tiled_partition<>()`
- Size of tile is template parameter

→ Known at compile time! Optimizations possible!

- Returns `thread_block_tile` object with additional member functions
  - `.shfl()`, `.shfl_down()`, `.shfl_up()`, `.shfl_xor()`
  - `.any()`, `.all()`, `.ballot()`; `.match_any()`, `.match_all()`

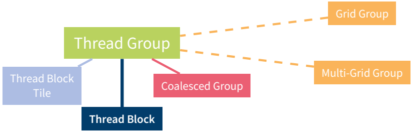
  → Intrinsic functions to work with threads inside a warp *(more later)*
- Example

```
cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cg::this_thread_block());
cg::thread_block_tile<4>  tile4  = cg::tiled_partition<4> (tile32);
```

Cooperative Groups

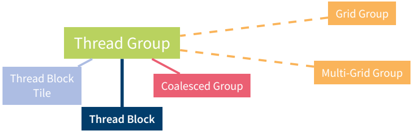
Coalesced Groups

Coalesced Group



- Get group of threads which is not diverged
- Threads have same state at point of API call
- `cg::coalesced_group active_threads = cg::coalesced_threads();`

Coalesced Group



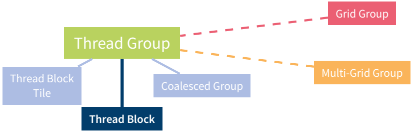
- Get group of threads which is not diverged
- Threads have same state at point of API call
- `cg::coalesced_group active_threads = cg::coalesced_threads();`
- Example

```
cg::coalesced_group active_threads = cg::coalesced_threads();
if (...) {
    cg::coalesced_group if_true_threads = cg::coalesced_threads();
    int rank = if_true_threads.thread_rank();
    cg::thread_group partition = cg::tiled_partition(if_true_threads, 2);
}
```

Cooperative Groups

Larger Groups

Grid Group

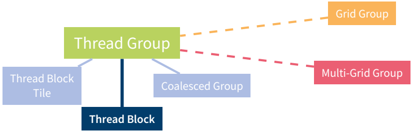


- Grid of blocks can also be entity now
- Synchronize across all blocks:

```
cg::grid_group grid = cg::this_grid();
grid.sync();
```

- Condition
  1. Blocks must be co-resident on device (Occupancy Calculator)
  2. Kernel must be launched with Cooperative Launch API
     `cudaLaunchCooperativeKernel()` instead of `<<<,>>>` syntax

Multi-Grid Group



- Group of blocks across multiple devices
- Synchronize blocks across devices:

```
cg::multi_grid_group multi_grid = cg::this_multi_grid();
multi_grid.sync();
```

- Condition
  1. Kernel must be launched with Cooperative Launch API `cudaLaunchCooperativeKernelMultiDevice()` instead of `<<<,>>>` syntax
     deprecated!
  2. Supported by architecture

- From here on: Optional!
- Overview
    - Task 2: Statically-tiled groups, atomic operations
    - Atomic operation
    - Warp-synchronous programming, warp intrinsics
    - Task 3: Warp-level reduction

→ Conclusions

Cooperative Groups with Tiled Partitions

TASK 2

Sub-divisions

- Location of code: `8-Cooperative_Groups/exercises/tasks/task2`
- See `Instructions.md` for explanations
- Follow TODOs to tile a CG and use kernel from Task 1; atomic operations needed
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on Cooperative Groups

Cooperative Groups with Tiled Partitions

TASK 2

Sub-divisions

- Location of code: `8-Cooperative_Groups/exercises/tasks/task2`
- See `Instructions.md` for explanations
- Follow TODOs to tile a CG and use kernel from Task 1; atomic operations needed
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on Cooperative Groups

Cooperative Groups with Tiled Partitions

TASK 2

Sub-divisions

- Location of code: `8-Cooperative_Groups/exercises/tasks/task2`
- See `Instructions.md` for explanations
- Follow TODOs to tile a CG and use kernel from Task 1; atomic operations needed — **Aside!**
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on Cooperative Groups

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Aside: Atomic Operations

Motivation

- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

Aside: Atomic Operations

Motivation

- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

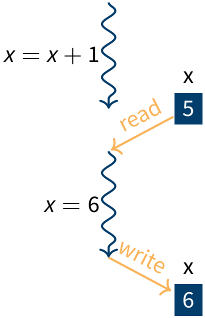
```
array[1] = array[1] + myvalue
```

Aside: Atomic Operations

Motivation

- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

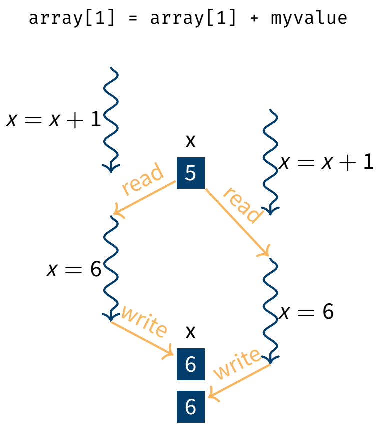
`array[1] = array[1] + myvalue`



Aside: Atomic Operations

Motivation

- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

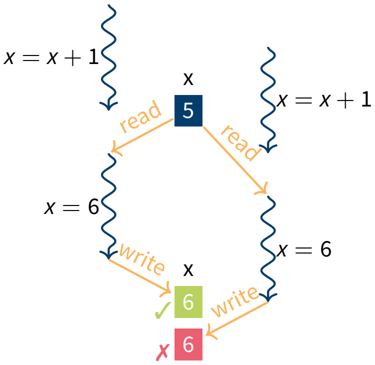


Aside: Atomic Operations

Motivation

- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

`array[1] = array[1] + myvalue`



Aside: Atomic Operations

Motivation

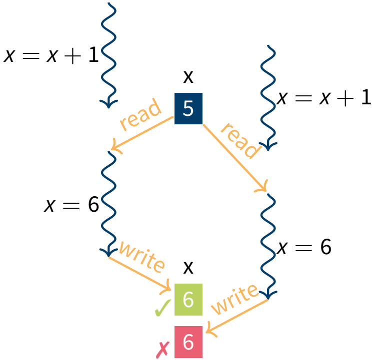
- Order execution of CUDA threads non-deterministic
- No problem, if each thread works on distinct data element
- What, if threads collaborate and share data? Read/Write to same element?

→ Atomic operations

  - Safe way to read and write to memory position by different threads
  - Data in global or shared memory
  - Example: `atomicAdd(&array[i], myvalue)`

- See CUDA Documentation

`array[1] = array[1] + myvalue`



Aside: Atomic Operations

Examples

- First argument to function (always): `address` of a value to potentially change
- Old value of `address` usually returned
- `int atomicOp(int * removeVal, int myVal)`

Aside: Atomic Operations

Examples

- First argument to function (always): `address` of a value to potentially change
- Old value of `address` usually returned
- `int atomicOp(int * removeVal, int myVal)`
- Examples

`atomicAdd(int* address, int val)` Add `val` to the value at `address`

`atomicExch(int* address, int val)` Store `val` at `address` location; return old value

`atomicMin(int* address, int val)` Store the minimum of `val` and the value at `address` at `address` location; return old value

`atomicCAS(int* address, int compare, int val)` The value at `address` is compared to `compare`. If true, `val` is stored at `address`; if false, the old value at `address` is stored. The old value at `address` is returned. Basic function: Compare And Swap

Cooperative Groups with Tiled Partitions

TASK 2

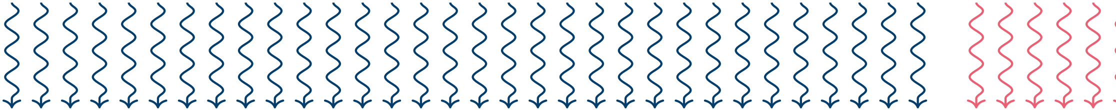
Sub-divisions

- Location of code: `8-Cooperative_Groups/exercises/tasks/task2`
- See `Instructions.md` for explanations
- Follow TODOs to tile a CG and use kernel from Task 1; atomic operations needed
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on Cooperative Groups

Warp-Synchronous Programming

Warp-Level Intrinsics

- Smallest set of executed threads: Warp
- Warp: 32 threads executed in SIMT/SIMD fashion
- Exchange data between threads of warp
  - Global memory: Slow
  - Shared memory: Faster
  - Directly (registers): Even faster
- Safe method access without race conditions
  - Global/shared memory: Atomic operations
  - Registers: **Warp-aggregated Atomic operations**



Warp Intrinsics Overview

`shfl(int lane)` Copy data from a target warp lane; also: other flavors (next slide)

Warp Intrinsics Overview

`shfl(int lane)`	Copy data from a target warp lane; also: other flavors (next slide)
`all(int pred)`	If predicate (*comparison, relation*) evaluates to non-zero (*true*) for all threads, return non-zero (*true*)
`any(int pred)`	If predicate evaluates to non-zero for any thread, return non-zero
`ballot(int pred)`	Return a bit mask which has 1s set for all thread for which predicate evaluates to non-zero

Warp Intrinsics Overview

`shfl(int lane)`	Copy data from a target warp lane; also: other flavors (next slide)
`all(int pred)`	If predicate (*comparison, relation*) evaluates to non-zero (*true*) for all threads, return non-zero (*true*)
`any(int pred)`	If predicate evaluates to non-zero for any thread, return non-zero
`ballot(int pred)`	Return a bit mask which has 1s set for all thread for which predicate evaluates to non-zero
`match_any(T value)`	Return a bit mask of threads which have same value of `value` as current thread; also: `match_all(T value)`

Warp Intrinsics Overview

`shfl(int lane)`	Copy data from a target warp lane; also: other flavors (next slide)
`all(int pred)`	If predicate (*comparison, relation*) evaluates to non-zero (*true*) for all threads, return non-zero (*true*)
`any(int pred)`	If predicate evaluates to non-zero for any thread, return non-zero
`ballot(int pred)`	Return a bit mask which has 1s set for all thread for which predicate evaluates to non-zero
`match_any(T value)`	Return a bit mask of threads which have same value of `value` as current thread; also: `match_all(T value)`

- Available as global device functions, with additional selection *mask* as first element (as `__shufl_sync()` etc.)
- Available as **member functions** of a `cg::tiled_partition` group (as `g.shfl()` etc.)
- Intrinsics automatically synchronize after operation – new since CUDA 9
- Value can only be retrieved if targeted lane also invokes intrinsic
- Per clock cycle: 32 shuffle instructions per SM $\rightarrow$ **very fast!**

Warp Intrinsic Example

Everyday I'm Shuffeling

- `shfl()`: Copy data from target warp lane
- Different flavors

`shfl()`	Copy data from warp lane with ID directly
`shfl_up()`	Copy data from relative warp lane with lower ID (shuffle *upstream*)
`shfl_down()`	Copy data from relative warp lane with higher ID (shuffle *downstream*)
`shfl_xor()`	Copy data from relative warp lane with ID as calculated by a bitwise XOR

- **Example: `shfl_down(value, N)` with `N` = 16, 8, …**

Transform Kernel to Warp-Level Reduction without Shared Memory

TASK 3

Expert level 11

- Location of code: `8-Cooperative_Groups/exercises/tasks/task3`
- See `Instructions.md` for explanations
- Follow TODOs to modify `maxKernel()` such that it uses warp-level atomic operations (and no shared memory)
- Compile with `make`, submit to batch system with `make run`
- See also CUDA C programming guide for details on warp-level functions

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Conclusions

Conclusions

- **CG** alternative model to create groups
- Groups are **entities**, have member functions
- Synchronizing is important (not mentioned before: `__syncwarps()`)
- **Warp-level functions** easily accessible from groups
- CG are quite new, let's see how they develop
- See also further literature in Appendix

Conclusions

- **CG** alternative model to create groups
- Groups are **entities**, have member functions
- Synchronizing is important (not mentioned before: `__syncwarps()`)
- **Warp-level functions** easily accessible from groups
- CG are quite new, let's see how they develop
- See also further literature in Appendix

Thank you
for your attention!

a.herten@fz-juelich.de

Appendix

Appendix

Further Literature

Glossary

References: Images

Further Literature

- NVIDIA Developer Blog: Cooperative Groups: Flexible CUDA Thread Programming
- NVIDIA Developer Blog: Inside Volta: The World's Most Advanced Data Center GPU
- NVIDIA Developer Blog: Using CUDA Warp-Level Primitives
- Talk at GPU Technology Conference 2018: Cooperative Groups by Kyrylo Perelygin and Yuan Lin
- Talk: Warp-synchronous programming with Cooperative Groups by Sylvain Collange
- Book: CUDA Programming by Shane Cook

Glossary I

API A programmatic interface to software by well-defined functions. Short for application programming interface. 41, 42, 44, 45

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 5, 35, 47, 48, 49, 50, 51, 52, 53, 54, 55, 58, 61, 62, 63, 64, 66

NVIDIA US technology company creating GPUs. 73

CG Cooperative Groups. 7, 8, 15, 35, 47, 48, 49, 58, 68, 69

GPU Graphics Processing Unit. 73

SIMD Single Instruction, Multiple Data. 60

SIMT Single Instruction, Multiple Threads. 5, 60

SM Streaming Multiprocessor. 61, 62, 63, 64

References: Images, Graphics I

[1] Yuriy Rzhemovskiy. *Teenage Penguins*. Freely available at Unsplash. URL: https://unsplash.com/photos/qFxS5FkUSAQ.

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE