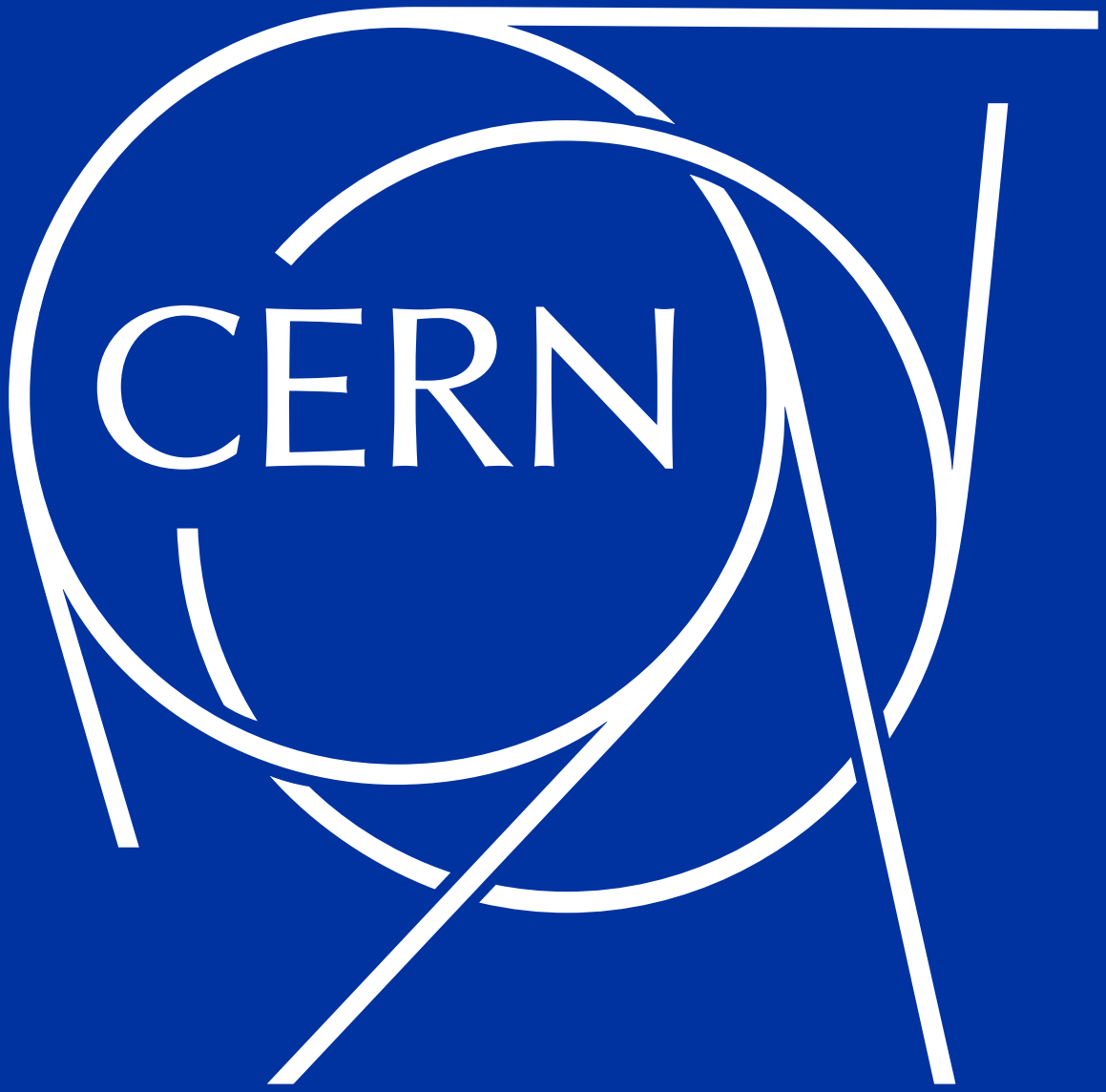

CERN

C++ Atomics

An Overview: Back to Basics

Abhishek Lekshmanan -(IT-ST-PDS)

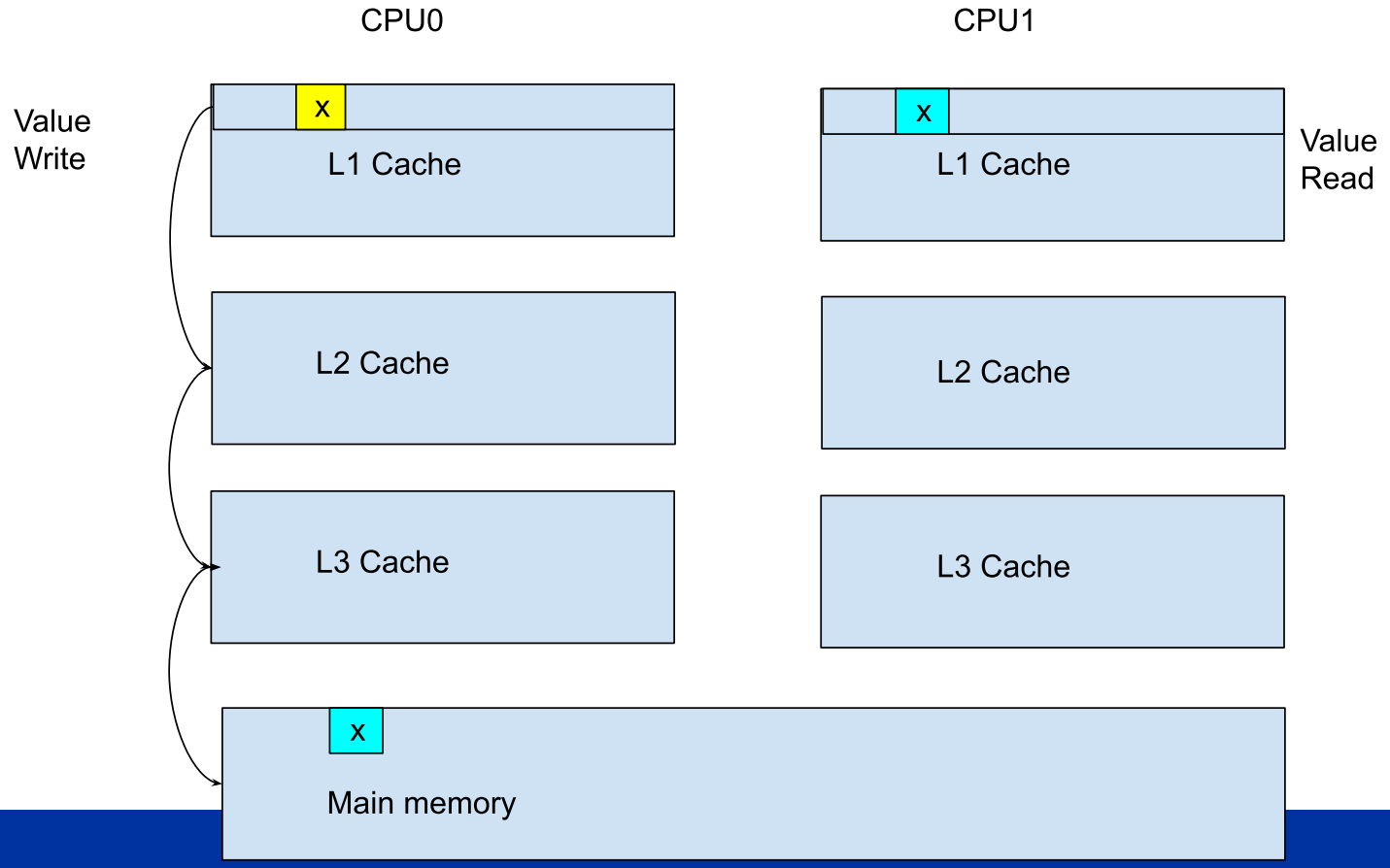
Objectives

- Since multi-threading concepts are so heavily used, just meant as a concepts refresher
- Multi threaded programming is hard, however the fundamental concepts used translate well into distributed systems programming nicely
- Building blocks for lock free data structures
- Checkout the talk later on managing locks on CERNBox & EOS later in the conf for a more practical example

Atomicity

- In concurrent programming, an atomic operation is an operation seen as non interruptible by other threads and appears as one single transaction
- Real life examples
  - Database transactions
  - Finance
  - Double Checked Locking Pattern (DCLP)
  - Always entirely successful or rollback semantics, no interruptions

Typical Multi CPU Arch



std::atomic

- Introduced in C++11
- Promises atomic operations on types
- Portable across platforms
- Can be applied on trivially_copyable and Copy Constructible/Assignable types.
- Default specialization for all integral types

```
std::atomic<int>;
struct Point { uint64_t x; uint64_t y };
struct d3 { uint64_t x, uint64_t y, uint64_t z};

// Since C++17 this can be checked at compile time
static_assert(std::atomic<Point>::is_always_lock_free);
static_assert(!std::atomic<d3>::is_always_lock_free);

// Runtime checking was always possible since C++11
std::atomic<Point>{}.is_lock_free();
std::atomic<d3>{}.is_lock_free();
```

std::atomic

- Only atomic operations are allowed on atomic types, compile time check!

```
i++; // OK
i += 1; // OK
i *= 2; // will not compile, no atomic mult instruction
```

```
i = i+1;
i = i*2;
```

std::atomic

- Only atomic operations are allowed on atomic types, compile time check!

```
i++; // OK
i += 1; // OK
i *= 2; // will not compile, no atomic mult instruction
```

BUT....

```
i = i+1;
i = i*2;
```

std::atomic

- Only atomic operations are allowed on atomic types, compile time check!

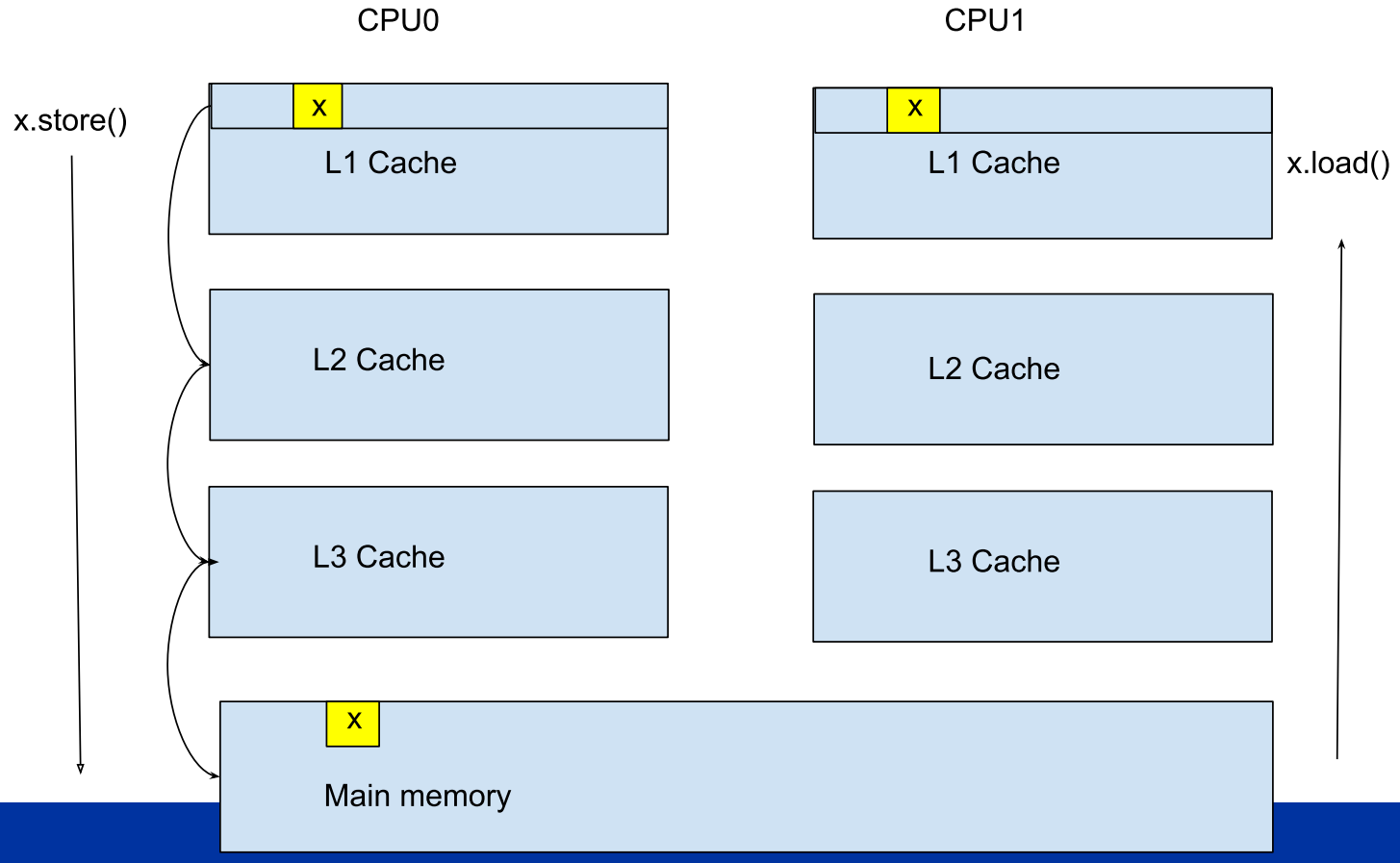
```
i++; // OK
i += 1; // OK
i *= 2; // will not compile, no atomic mult instruction
```

BUT....

```
i = i+1;
i = i*2;
```

atomic load and store are 2 different transactions unless you use the various cas/fetch method or the correct operator overloads, the transactions aren't atomic

CPU view with atomics



Operations on atomic

Operation	Bool	Integer	Generic Ptr
Load/Store	x	x	x
Exchange	x	x	x
CAS (compare/exchange)	x	x	x
Fetch-Add/Sub		x	x
AND		x	
OR		x	
XOR		x	

Operations on atomic

- Only these transactions are guaranteed to be a single atomic transaction
- Operator overloads exist, however almost always it is better to be explicit and use member functions
- Compare And Swap is the building block for atomic transactions, and can be used to do almost any operation atomically

```
std::atomic<int> i;
// Thread 1;
int local_i = i;
while (!i.compare_and_exchange_strong(local_i, i*2)) {}
```

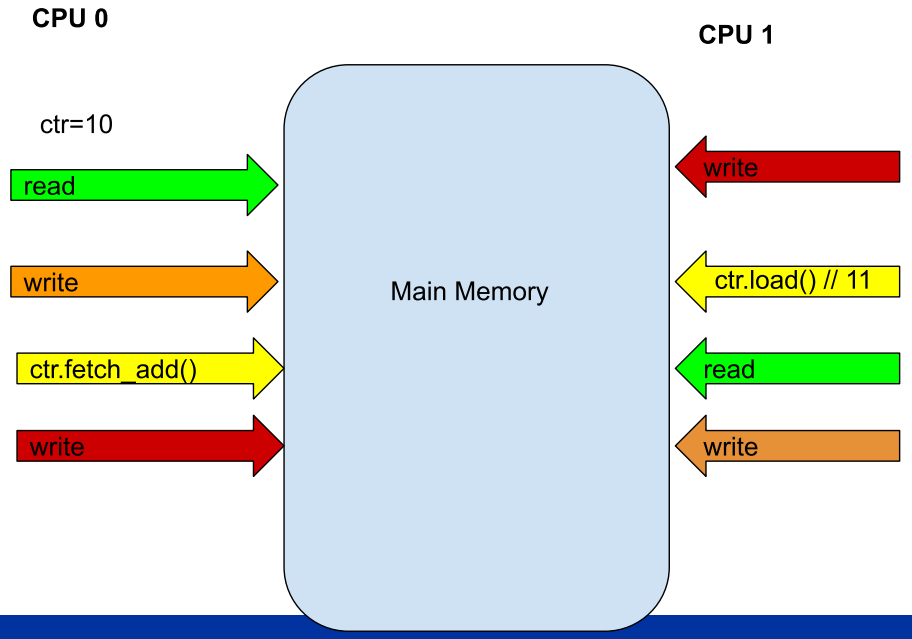
- All operations support a memory_order flag. Operator overloads assume default memory order (std::memory_order_seq_cst).

Memory Barriers

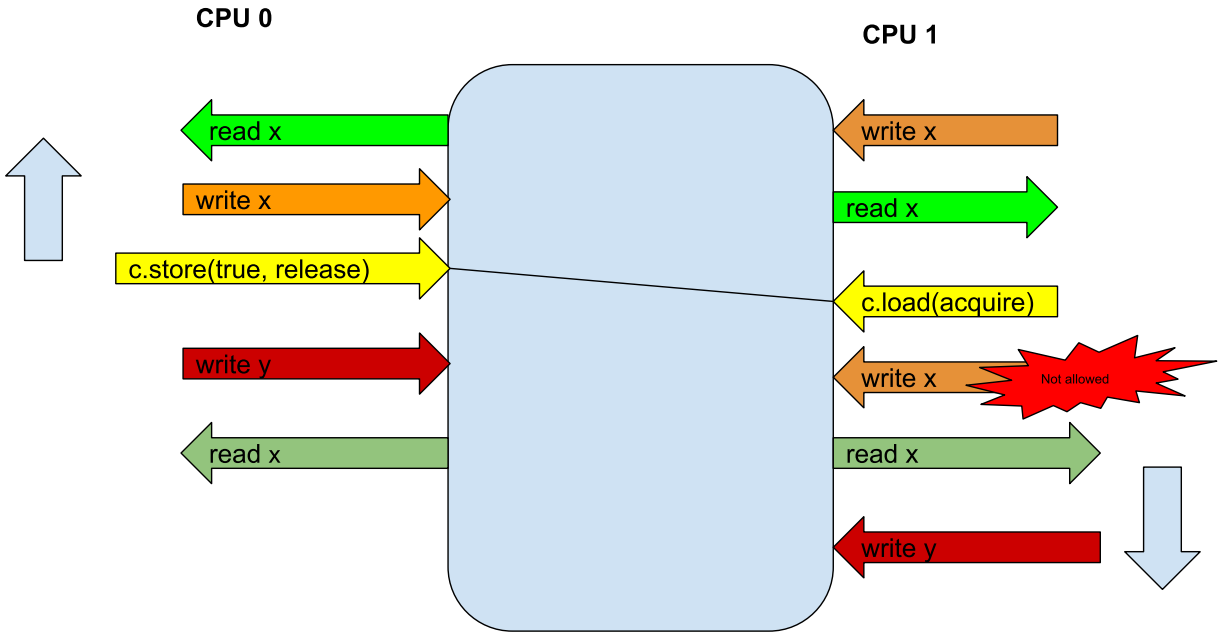
- While the simple CPU view depicts what happens to an atomic variable itself, this is not the full picture
- Atomics are your building blocks in revealing memory to other threads;
- This is achieved by memory barriers; pre C++11 there were no guarantees you'd have to write asm/intrinsic yourself
- C++11 provides ~3 different worlds of memory orders living in harmony, broadly **memory_order_relaxed**, **memory_order_acq_rel**, **memory_order_seq_cst** (omiting consume ordering)*
- Typically lock-free/wait-free data structures build upon a RCU pattern; for eg LL with head change as CAS

Memory Order Relaxed

Only guarantees on the atomic variable itself, no guarantees on any surrounding writes and reads



Memory Order Acquire Release



Acquire Release: details

- All memory writes (both non atomic & atomic) *before* a store with **memory_order_release** in program order shall be visible on another thread with a **memory_order_acquire**
- Vice versa for happens after with acquire
- Half memory barriers, Order guarantees only affect cooperating threads
- Mutexes/spinlocks under the hood are *at least* an acquire operation for a lock and release operation with unlock, ensuring whatever happens in a critical section is seen by another thread entering the cs.
- Useful building block for lock free data structures, inter-thread ordering provided

Memory Order Sequential Consistency

- **Bidirectional barrier**: Default flag assumed on all atomic operations unless explicitly specified otherwise
- **Global program order**: Establishes strong happens before and after consistency at a global level
- **Expensive**: Orders of magnitudes slower
- Makes it easier to reason esp when multiple atomics are involved, but may not be necessary. for eg. lock impl doesn't need seq_cst

Benchmark Run single int store() 8x4.6 GHz CPUs	Time	CPU	freq
BM_memory_order_seq_cst/real_time/threads:1	443 ns	443 ns	225.746M/s
BM_memory_order_seq_cst/real_time/threads:2	231 ns	461 ns	432.928M/s
BM_memory_order_seq_cst/real_time/threads:4	123 ns	492 ns	811.682M/s
BM_memory_order_seq_cst/real_time/threads:8	118 ns	924 ns	850.25M/s
BM_memory_order_release/real_time/threads:1	28.7 ns	28.6 ns	3.48057G/s
BM_memory_order_release/real_time/threads:2	15.1 ns	30.2 ns	6.62921G/s
BM_memory_order_release/real_time/threads:4	8.59 ns	34.3 ns	11.6379G/s
BM_memory_order_release/real_time/threads:8	5.36 ns	40.0 ns	18.6547G/s

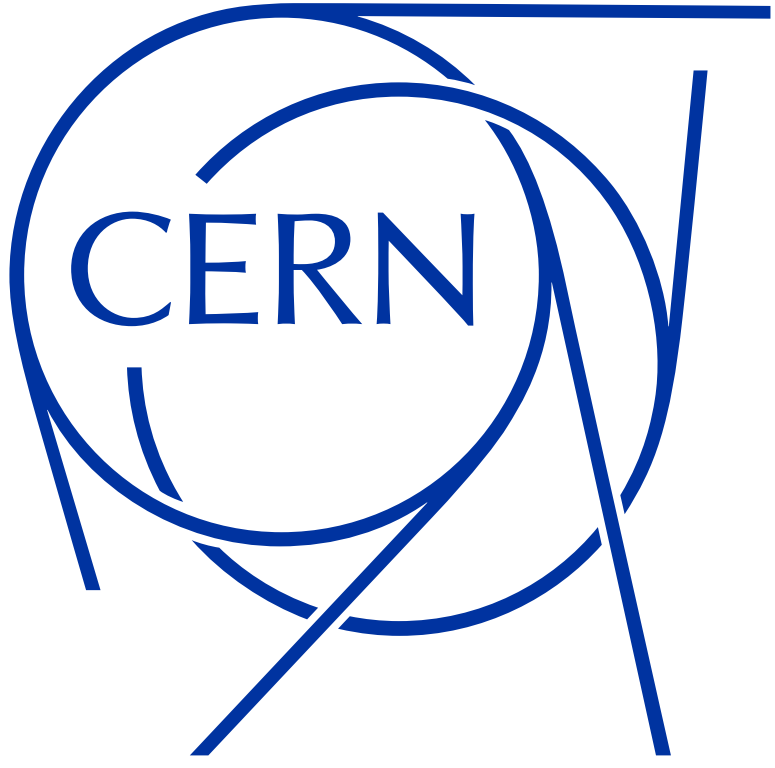
Takeaways

- Atomic guarantees data consistency in concurrent contexts
- CAS and memory orders are the building blocks to designing lock free structures
- Be careful when mixing atomic & non-atomic operations
- Concurrency is hard, if you're reaching out to atomic to build data structures for performance, go all the way!
    - Memory orders expresses what you want the hardware to do and easier to reason
    - Choice of memory barrier can affect performance, also affects platform runtimes

References

- PIKUS, F. G. (2021). Threads, Memory and Concurrency. In Art of writing efficient programs; PACKT PUBLISHING LIMITED.
- Fedor Pikus: CppCon 2017 Talk: C++ Atomics from basic to advanced
- Michael Wong: CppCon 2015 Talk: C++11/14/17 atomics & memory model
- Paul E McKenney: CppCon 2015 Talk: C++ Atomics: The sad story of memory_order_consume
- Olivier Giroux: CppCon 2019 Talk: The One Decade Task: Puting std::atomic in CUDA

Questions?



www.cern.ch

Benchmark source file

```
#include <atomic>
#include "benchmark/benchmark.h"

static void BM_memory_order_seq_cst(benchmark::State& state)
{
  std::atomic<int> x{0};
  for (auto _ :state) {
    for (int i=0;i<100;i++)
      x.store(1);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(100*state.iterations());
}

static void BM_memory_order_release(benchmark::State& state)
{
  std::atomic<int> x{0};
  for (auto _ :state) {
    for (int i=0;i<100;i++)
      x.store(1, std::memory_order_release);
    benchmark::ClobberMemory();
  }
  state.SetItemsProcessed(100*state.iterations());
}

BENCHMARK(BM_memory_order_seq_cst)->ThreadRange(1,8)->UseRealTime();
BENCHMARK(BM_memory_order_release)->ThreadRange(1,8)->UseRealTime();
BENCHMARK_MAIN();
```