THE CUDA C++ STANDARD LIBRARY

Bryce Adelstein Lelbach

CUDA C++ Core Libraries Lead

ISO C++ Library Evolution Chair, US Programming Languages Chair

@blelbach

"[CUDA C++] lets you use the powerful [ISO] C++ programming language to develop high performance algorithms accelerated by thousands of parallel threads running on GPUs."

— *An Even Easier Introduction to CUDA*, Parallel For All Blog, January 2017

CUDA C++ Is A Superset of ISO C++

Host processors can use **alone**	All processors can use **isolated**	All processors can use **together**

`throw`
`catch`
`typeid`
`dynamic_cast`
`thread_local`
`std::`
	`virtual` functions
	
	function pointers
	
	lambdas
		<rest of ISO C++>

ISO C++ == Core Language + Standard Library

ISO C++ == Core Language + Standard Library

C++ without a Standard Library is severely diminished.

CUDA C++ == Core Language + ???

CUDA C++ Needs a C++ Standard Library

Hardware Feature Exposure: Standard library abstractions are key to GPU hardware feature exposure; bugs & limitations can be overcome in the abstractions.

Productivity & Performance: CUDA C++ programmers waste time re-implementing standard library facilities; they often encounter performance & correctness pitfalls when doing so.

Consistency & Interoperability: Duplicated re-implementations of standard library facilities across the CUDA ecosystem lack consistency and actively inhibit interoperability.

CUDA C++ == Core Language + **libcu++**

Version 1 in CUDA 10.2!

libcu++ is the

opt-in,

heterogeneous,

incremental

CUDA C++ Standard Library.

Opt-in

Does not interfere with or replace your host standard library.

`#include <…>` `std::`	ISO C++, `__host__` only. Strictly conforming to ISO C++.
`#include <cuda/std/…>` `cuda::std::`	CUDA C++, `__host__ __device__`. Strictly conforming to ISO C++.
`#include <cuda/…>` `cuda::`	CUDA C++, `__host__ __device__`. Conforming extensions to ISO C++.

NVIDIA

Opt-in

Does not interfere with or replace your host standard library.

```
#include <atomic>
std::atomic<int> x;

#include <cuda/std/atomic>
cuda::std::atomic<int> x;

#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

Heterogeneous

Copyable/Movable objects can migrate between host & device.

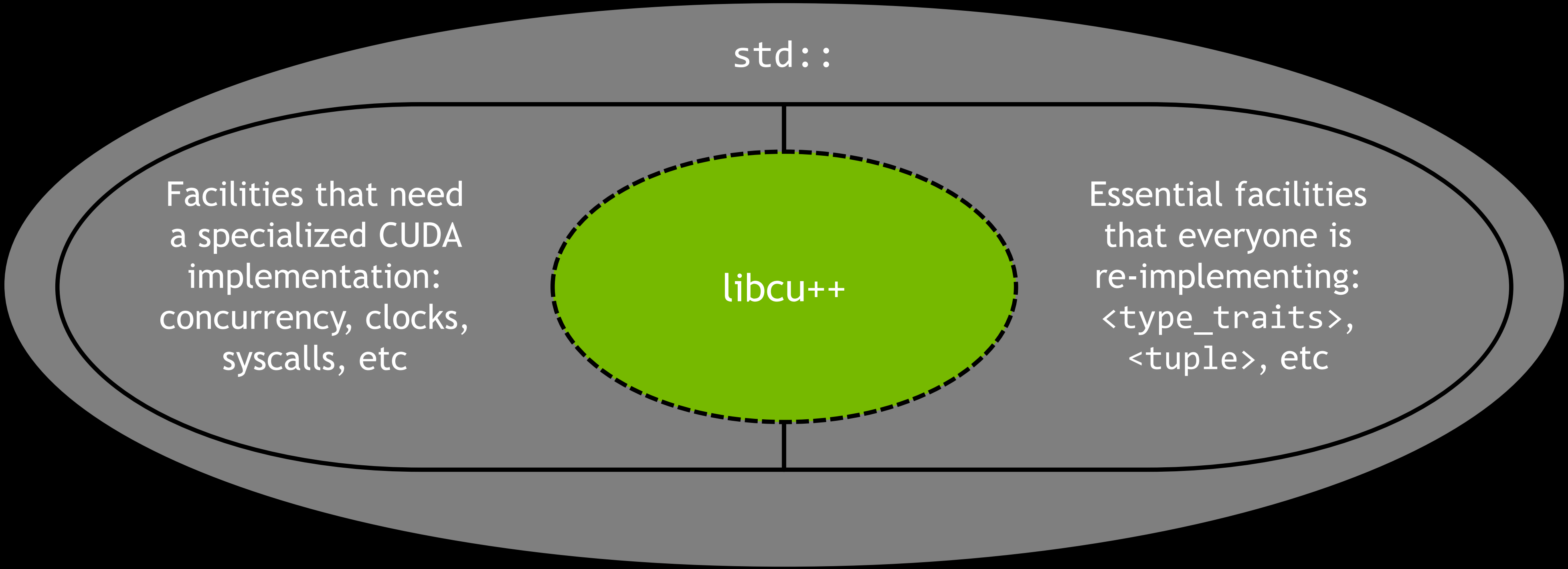
Host & device can call all (member) functions.

Host & device can concurrently use synchronization primitives*.

*: Synchronization primitives must be in managed memory and be declared with `cuda::std::thread_scope_system`.

Incremental

Not a complete standard library today; each release will add more.



Based on LLVM’s libc++

Forked from LLVM’s libc++.

License: Apache 2.0 with LLVM Exception.

NVIDIA is already contributing back to the community:

- Freestanding `atomic<T>`: reviews.llvm.org/D56913
- C++20 synchronization library: reviews.llvm.org/D68480

libcu++ Release Schedule

Version 1 (CUDA 10.2, now): `<atomic>` (Pascal+), `<type_traits>`.

Version 2 (CUDA next): `atomic<T>::wait/notify`, `<barrier>`, `<latch>`, `<counting_semaphore>` (all Volta+), `<chrono>`, `<ratio>`, `<functional>` minus `function`.

Future priorities: `atomic_ref<T>`, `<complex>`, `<tuple>`, `<array>`, `<utility>`, `<cmath>`, string processing, …

```
namespace cuda {

enum thread_scope {
  thread_scope_system, // All threads.
  thread_scope_device,
  thread_scope_block
};

template <typename T,
          thread_scope S = thread_scope_system>
struct atomic;

namespace std {
  template <typename T>
  using atomic = cuda::atomic<T>;
} // namespace std

} // namespace cuda
```

```
__host__ __device__
void signal_flag(volatile int& flag) {
  // ^^^ volatile was "notionally right" in legacy CUDA C++.

  // vvv "Works" for a store but is UB (volatile != atomic).
  flag = 1;
}
```

```
__host__ __device__
void signal_flag(volatile int& flag) {
  // ^^^ volatile was "notionally right" for flag in legacy CUDA C++.
  __threadfence_system(); // <- Should be fused on the operation.
  // vvv "Works" for a store but is UB (volatile != atomic).
  flag = 1;
}
```

```
__host__ __device__
void signal_flag(volatile int& flag) {
  // ^^^ volatile was "notionally right" for flag in legacy CUDA C++.
  __threadfence_system(); // <- Should be fused on the operation.
  // vvv We "cast away" the `volatile` qualifier.
  atomicExch((int*)&flag, 1); // <- Ideally want an atomic store.
}
```

```
__host__ __device__
void signal_flag_better(atomic<bool>& flag) {
  flag = true;
}
```

```
__host__ __device__
void signal_flag_even_better(atomic<bool>& flag) {
  flag.store(true, memory_order_release);
}
```

```
__host__ __device__
void signal_flag_excellent(atomic<bool>& flag) {
  flag.store(true, memory_order_release);
  flag.notify_all(); // <- Will make sense later (Version 2).
}
```

```
__host__ __device__
int poll_flag_then_read(volatile int& flag, volatile int& data) {
  // ^^^ volatile was "notionally right" in legacy CUDA C++.
  // vvv “Works" but is UB (volatile != atomic).
  while (1 != flag)
    ; // <- Spinloop without backoff is bad under contention.

  return data;
}
```

```
__host__ __device__
int poll_flag_then_read(volatile int& flag, int& data) {
  // ^^^ volatile was "notionally right" in legacy CUDA C++.
  // vvv "Works" but is UB (volatile != atomic).
  while (1 != flag)
    ; // <- Spinloop without backoff is bad under contention.
  __threadfence_system(); // <- 9 out of 10 of you forget this one!
  return data;
}
```

```
__host__ __device__
int poll_flag_then_read(volatile int& flag, volatile int& data) {
  // ^^^ volatile was "notionally right" in legacy CUDA C++.
  // vvv We "cast away" the volatile qualifier.
  while (1 != atomicAdd((int*)&flag, 0)) // <- Should be atomic load.
    ; // <- Spinloop without backoff is bad under contention.
  __threadfence_system(); // <- 9 out of 10 of you forget this one!
  return data;
}
```

```
__host__ __device__
int poll_flag_then_read_better(atomic<bool>& flag, int& data) {
  while (!flag)
    ; // <- Spinloop without backoff is bad under contention.
  return data;
}
```

```
__host__ __device__
int poll_flag_then_read_even_better(atomic<bool>& flag, int& data) {
  while (!flag.load(memory_order_acquire))
    ; // <- Spinloop without backoff is bad under contention.
  return data;
}
```

```
__host__ __device__
int poll_flag_then_read_excellent(atomic<bool>& flag, int& data) {
  flag.wait(false, memory_order_acquire); // Version 2.
  // ^^^ Backoff to mitigate heavy contention.
  return data;
}
```

```
// Mixing scopes can be a messy error; we prevent it at compile time.
__host__ __device__ void foo() {
  atomic<bool> s_flag;
  signal_flag(s_flag); // Ok; expects and got system atomic type.

  atomic<bool, thread_scope_device> d_flag;
  signal_flag(d_flag); // Compile error; expects system atomic type.
}
```

```
// Writing __host__ __device__ functions today is nearly impossible.
__host__ __device__ void bar(volatile int& a) {
  #ifdef __CUDA_ARCH__
    atomicAdd((int*)&a, 1);
  #else
    // What do I write here for all the CPUs & compilers I support?
  #endif
}
```

```
__host__ __device__ void bar_better(atomic<int>& a) {
  a.fetch_add(1, memory_order_relaxed);
}
```

Stop Using Legacy CUDA Atomics (`atomic[A-Z]*`):

Sequential consistency & acquire/release are not first-class.

Device-only.

Memory scope is a property of operations not objects.

Atomicity is a property of operations not objects.

Stop Using `volatile` for synchronization:

`volatile` != atomic.

`volatile` is a vague pact; `atomic<T>` has clear semantics.

Volta+ NVIDIA GPUs deliver and libcu++ exposes:

C++ Parallel Forward Progress Guarantees.

The C++ Memory Model.

Why does this matter?

Why does this matter?

Volta+ NVIDIA GPUs and libcu++ enable a wide range of concurrent algorithms & data structures previously unavailable on GPUs.

	No limitations on thread delays	Threads delayed infinitely often	Thread delays limited
Every thread makes progress	Wait-free	Obstruction-free	Starvation-free
Some thread makes progress	Lock-free	Clash-free	Deadlock-free
	Non-Blocking		Blocking

Source: http://www.cs.tau.ac.il/~shanir/progress.pdf

	No limitations on thread delays	Threads delayed infinitely often	Thread delays limited
Every thread makes progress	Wait-free	Obstruction-free	Starvation-free
Some thread makes progress	Lock-free	Clash-free	Deadlock-free
	Non-Blocking		Blocking

Weakly Parallel Forward Progress

Pre Volta NVIDIA GPUs
Other GPUs

Source: http://www.cs.tau.ac.il/~shanir/progress.pdf

	No limitations on thread delays	Threads delayed infinitely often	Thread delays limited
Every thread makes progress	Wait-free **Weakly Parallel Forward Progress**	Obstruction-free	Starvation-free **Parallel Forward Progress**
Some thread makes progress	Lock-free **Pre Volta NVIDIA GPUs** **Other GPUs**	Clash-free	Deadlock-free **Only Volta+**
	Non-Blocking		Blocking

Source: http://www.cs.tau.ac.il/~shanir/progress.pdf

Why does this matter?

Volta+ NVIDIA GPUs and libcu++ enable a wide range of concurrent algorithms & data structures previously unavailable on GPUs.

More concurrent algorithms & data structures means more code can run on GPUs!

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t                capacity_;
  Key*                    keys_;
  Value*                  values_;
  atomic<state_type>*     states_;
  Hash                    hash_;
  Equal                   equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t               capacity_;
  Key*                   keys_;
  Value*                 values_;
  atomic<state_type>*    states_;
  Hash                   hash_;
  Equal                  equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t                capacity_;
  Key*                    keys_;
  Value*                  values_;
  atomic<state_type>*     states_;
  Hash                    hash_;
  Equal                   equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t                capacity_;
  Key*                    keys_;
  Value*                  values_;
  atomic<state_type>*     states_;
  Hash                    hash_;
  Equal                   equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t              capacity_;
  Key*                  keys_;
  Value*                values_;
  atomic<state_type>*   states_;
  Hash                  hash_;
  Equal                 equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t             capacity_;
  Key*                 keys_;
  Value*               values_;
  atomic<state_type>*  states_;
  Hash                 hash_;
  Equal                equal_;
};
```

```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __host__ __device__ Value* try_insert(Key const& key, Value const& value);

private:
  uint64_t            capacity_;
  Key*                keys_;
  Value*              values_;
  atomic<state_type>* states_;
  Hash                hash_;
  Equal               equal_;
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    // ...
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```


NVIDIA.

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        // ...
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          // ...
        }
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.

          return values_ + index;
        }
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.

          return values_ + index;
        }
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      }
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      } // If we didn't fill the slot, wait for it to be filled and check if it matches.
      while (state_filled != states_[index].load(memory_order_acquire))
        ;
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      } // If we didn't fill the slot, wait for it to be filled and check if it matches.
      states_[index].wait(state_reserved, memory_order_acquire);
      // ...

    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      } // If we didn't fill the slot, wait for it to be filled and check if it matches.
      states_[index].wait(state_reserved, memory_order_acquire);
      if (equal_(keys_[index], key)) return values_ + index; // Someone else inserted.
      // ...
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      } // If we didn’t fill the slot, wait for it to be filled and check if it matches.
      states_[index].wait(state_reserved, memory_order_acquire);
      if (equal_(keys_[index], key)) return values_ + index; // Someone else inserted.
      index = (index + 1) % capacity_; // Collision: keys didn’t match. Try the next slot.
    }
    return nullptr; // If we are here, the container is full.
  }
};
```


NVIDIA

```
struct concurrent_insert_only_map {
  __host__ __device__ Value* try_insert(Key const& key, Value const& value) {
    auto index(hash_(key) % capacity_);
    for (uint64_t i = 0; i < capacity_; ++i) { // Linearly probe up to `capacity_` times.
      state_type old = states_[index].load(memory_order_acquire);
      while (old == state_empty) { // As long as the slot is empty, try to lock it.
        if (states_[index].compare_exchange_weak(old, state_reserved, memory_order_acq_rel)) {
          // We locked it by setting the state to `state_reserved`; now insert the key & value.
          new (keys_ + index) Key(key);
          new (values_ + index) Value(value);
          states_[index].store(state_filled, memory_order_release); // Unlock the slot.
          states_[index].notify_all(); // Wake up anyone who was waiting for us to fill the slot.
          return values_ + index;
        }
      } // If we didn’t fill the slot, wait for it to be filled and check if it matches.
      states_[index].wait(state_reserved, memory_order_acquire);
      if (equal_(keys_[index], key)) return values_ + index; // Someone else inserted.
      index = (index + 1) % capacity_; // Collision: keys didn’t match. Try the next slot.
    }
    return nullptr; // If we are here, the container is full.
  }
};
```

There’s a whole new world of algorithms and data structures that can be accelerated with Volta+ NVIDIA GPUs using libcu++ atomics.

libcu++

The CUDA C++ Standard Library

Opt-in, heterogeneous, incremental C++ standard library for CUDA.

Open source; port of LLVM's libc++; contributing upstream.

Version 1 (CUDA 10.2): `<atomic>` (Pascal+), `<type_traits>`.

Version 2 (CUDA next): `atomic<T>::wait/notify`, `<barrier>`, `<latch>`, `<counting_semaphore>` (all Volta+), `<chrono>`, `<ratio>`, `<functional>` minus function.

Future priorities: `atomic_ref<T>`, `<complex>`, `<tuple>`, `<array>`, `<utility>`, `<cmath>`, string processing, …

@blelbach