THE CUDA C++ STANDARD LIBRARY

Bryce Adelstein Lelbach
CUDA C++ Core Libraries Lead
ISO C++ Library Evolution Incubator Chair, ISO C++ Tooling Study Group Chair
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++ == Core Language + \texttt{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 <...>  // ISO C++, __host__ only.
std::          // Strictly conforming to ISO C++.

#include <cuda/std/...>  // CUDA C++, __host__ __device__.
cuda::std::        // Strictly conforming to ISO C++.

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

Copyright (C) 2019 NVIDIA
Incremental
Not a complete standard library today; each release will add more.

- Same Optimal Design for Host/Device (just add __device__) - std::min/std::max
- Different Optimal Design for Host/Device - std::hash
- Highest Impact
  - std::array
  - std::tuple
  - Already Available (in Thrust or CUB)
- Large Effort to Implement for Device
  - std::atomic
  - std::function
  - std::iostream
Incremental
Not a complete standard library today; each release will add more.

We are prioritizing facilities that:
- Are neither trivial nor difficult to port.
- Are not already available in CUDA.
- Have a high impact.

Already Available (in Thrust or CUB)

std::array
std::function
std::iostream
std::atomic
std::hash
std::min/std::max
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 (1H 2020): `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

#include <C++>
__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(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.
    flag = 1; // <- "Works" for a store but is UB (volatile != atomic).
}
__host__ __device__
void signal_flag_better(atomic<bool>& flag) {
  flag = true;
}
```cpp
__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, int& data) {
    // ^^^ volatile was "notionally right" for flag 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; // <- Even if volatile, you still need the fence.
}
__host__ __device__
int poll_flag_then_read(volatile int& flag, int& data) {
    // ^^^ volatile was "notionally right" for flag in legacy CUDA C++.
    // vvv We "cast away" the volatile qualifier.
    while (1 != flag) // <- “Works" but is UB (volatile != atomic).
        ; // <- Spinloop without backoff is bad under contention.
__threadfence_system(); // <- 9 out of 10 of you forget this one!
    return data; // <- Even if volatile, you still need the fence.
}
__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
}

#include <C++>
__host__ __device__ void bar_better(atomic<int>& a) {
    a += 1;
}

#include <C++>
__host__ __device__ void bar_even_better(atomic<int>& a) {
    a.fetch_add(1, memory_order_relaxed);
}
Stop Using Legacy 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 libc++++ exposes:

- C++ Parallel Forward Progress Guarantees.
- The C++ Memory Model.

Why does this matter?
Why does this matter?

Volta+ NVIDIA GPUs and libcub++ enable a wide range of concurrent algorithms & data structures previously unavailable on GPUs.
<table>
<thead>
<tr>
<th>Every thread makes progress</th>
<th>No limitations on thread delays</th>
<th>Threads delayed infinitely often</th>
<th>Thread delays limited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait-free</td>
<td>Obstruction-free</td>
<td>Starvation-free</td>
<td></td>
</tr>
<tr>
<td>Lock-free</td>
<td>Clash-free</td>
<td>Deadlock-free</td>
<td></td>
</tr>
</tbody>
</table>

Every thread makes progress

<table>
<thead>
<tr>
<th>No limitations on thread delays</th>
<th>Threads delayed infinitely often</th>
<th>Thread delays limited</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weakly Parallel Forward Progress</strong></td>
<td><strong>Pre Volta NVIDIA GPUs</strong></td>
<td><strong>Starvation-free</strong></td>
</tr>
<tr>
<td><strong>Non-Blocking</strong></td>
<td><strong>Deadlock-free</strong></td>
<td><strong>Blocking</strong></td>
</tr>
<tr>
<td><strong>Source:</strong> <a href="http://www.cs.tau.ac.il/~shanir/progress.pdf">http://www.cs.tau.ac.il/~shanir/progress.pdf</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why does this matter?

Volta+ NVIDIA GPUs and libcud++ 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_;      
};

#include <C++>
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.
  }
};
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.
        // ...
        // ...
      } // 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.
  }
};

#include <C++>
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;
        }
      }
    }
    // ...
  } // If we are here, the container is full.
};

#include <C++>
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.
};

#include <C++>
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); // ...
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.
    }
};

#include <C++>
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.
};

#include <C++>
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.
};
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 (next week): `<atomic>` (Pascal+), `<type_traits>`,

Version 2 (1H 2020): `atomic<T>::wait/notify`, `<barrier>`, `<latch>`, `<counting_semaphore>` (all Volta+), `<chrono>`, `<ratio>`, `<functional> minus function.

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

#include `<C++>`

Copyright (C) 2019 NVIDIA