THE NVIDIA C++ STANDARD LIBRARY

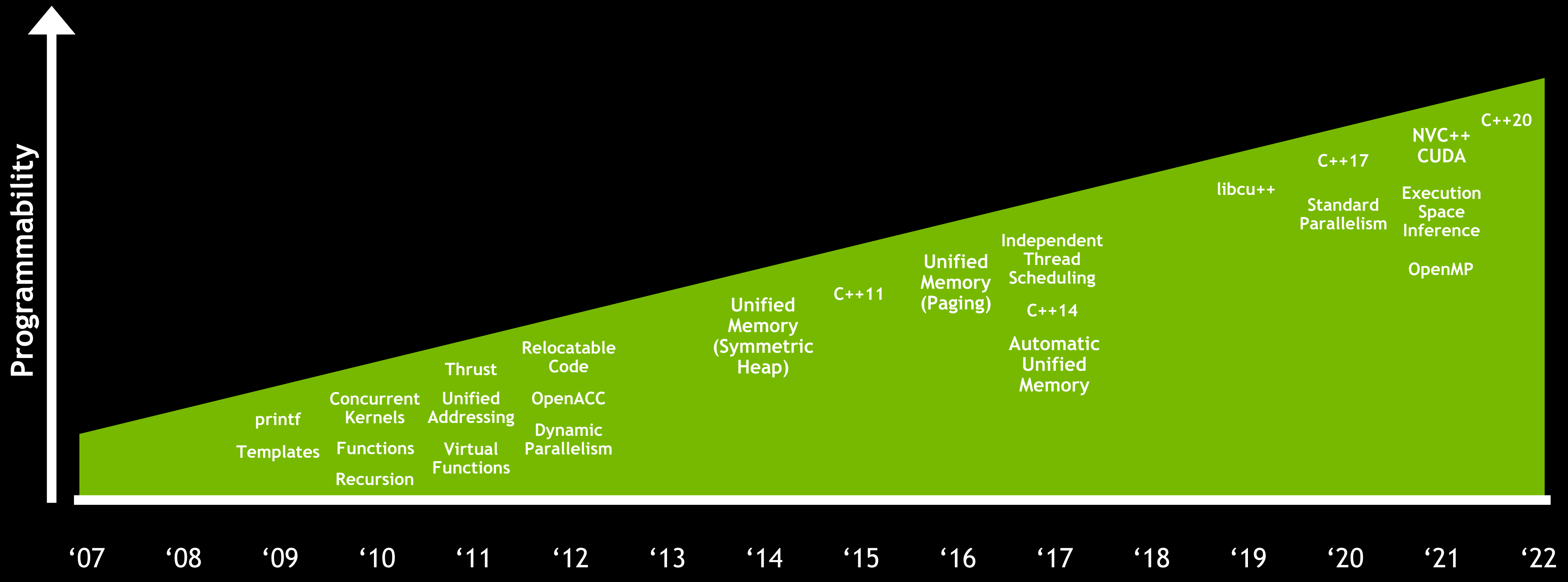
Bryce Adelstein Lelbach

HPC Programming Models Architect

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

@blelbach

Expanding GPUs Can Do

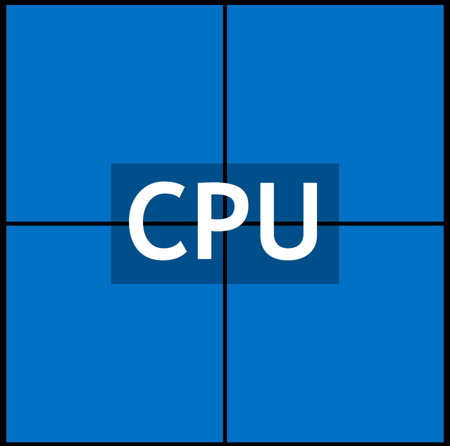


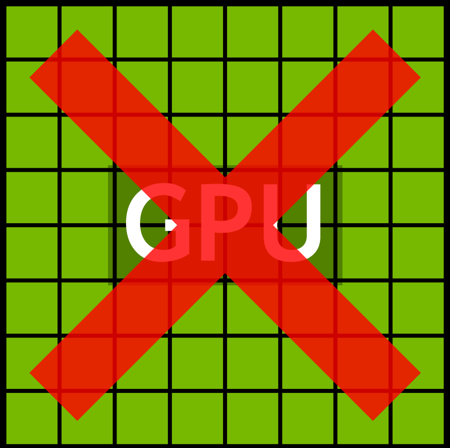
C++ == Core Language + Standard Library

== Core Language + Standard Library

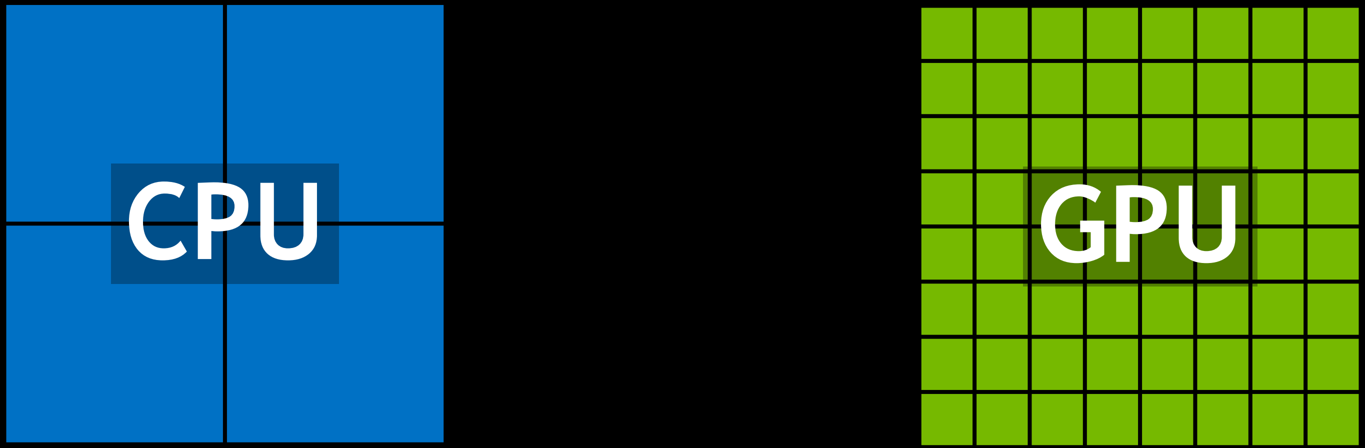
C++ without a Standard Library is severely diminished.

Other C++ Standard Library Implementations





The C++ Standard Library for Your Entire System



The C++ Standard Library for Your Entire System

https://github.com/NVIDIA/libcudacxx

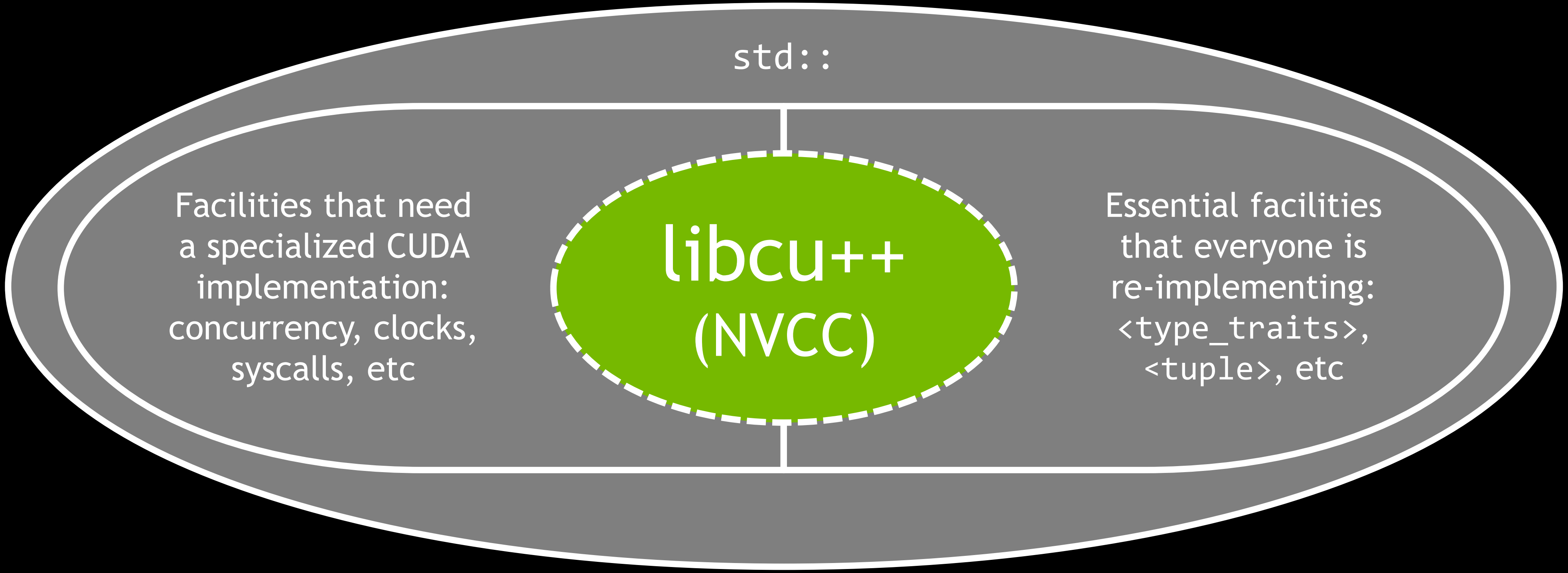
Host Compiler's Standard Library (GCC, MSVC, etc)

#include <…> std::	ISO C++, __host__ only. Complete, strictly conforming to Standard C++.

#include <cuda/std/…> cuda::std::	CUDA C++, __host__ __device__. Subset, strictly conforming to Standard C++.
#include <cuda/…> cuda::	CUDA C++, __host__ __device__. Conforming extensions to Standard C++.

libcu++ (NVCC)

libcu++ does not interfere with or replace your host Standard Library.



libcu++ is not a complete Standard Library today.
Each release adds more features.

The NVIDIA HPC Compiler

4 models

ISO Standard Parallelism

CUDA

OpenACC

OpenMP

3 languages

C++ C Fortran

2 targets

CPU

GPU

1 Compiler for Your Entire System

Learn More: [Inside NVC++ and NVFORTRAN, Bryce Adelstein Lelbach [S31358]]

NVIDIA

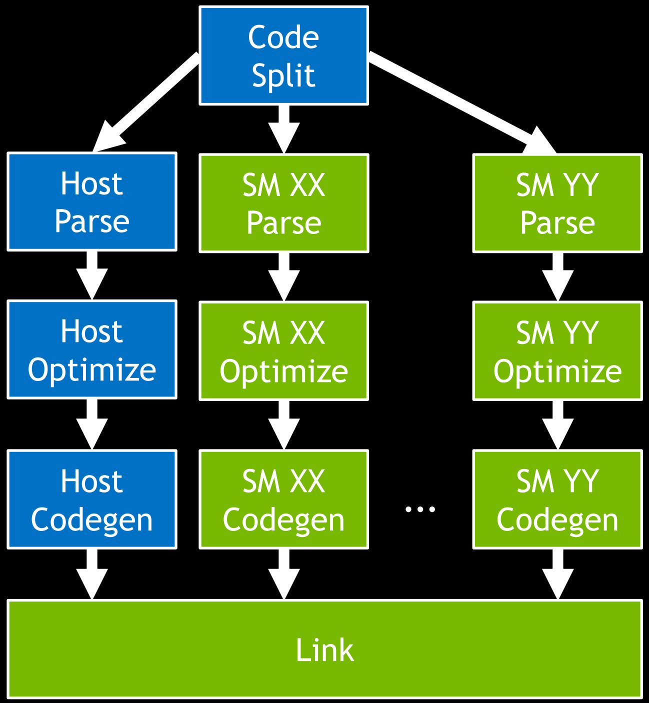
CUDA C++ for NVC++

Coming soon!

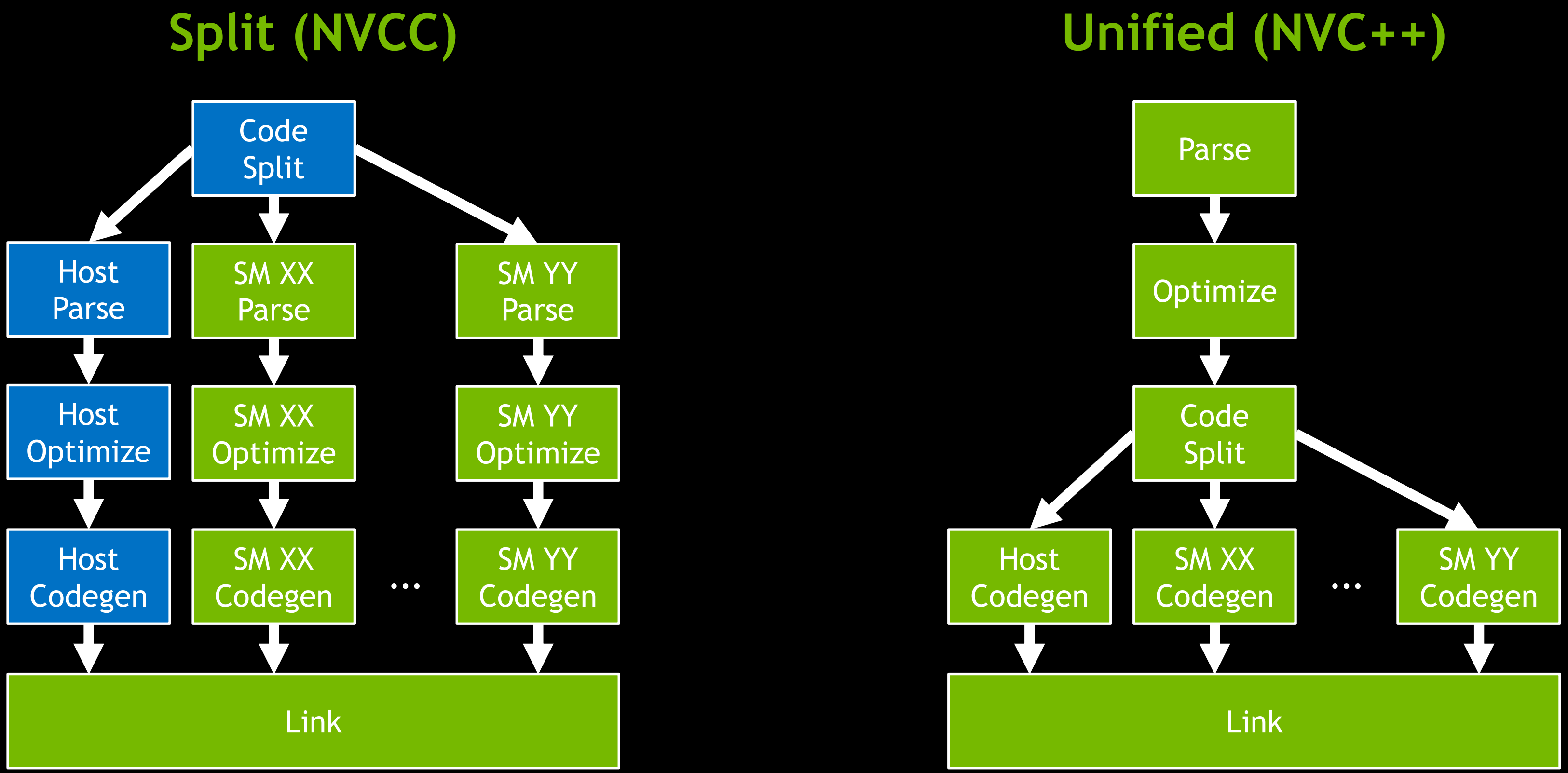
Learn More: Inside NVC++ and NVFORTRAN, Bryce Adelstein Lelbach [S31358]

Heterogeneous Compilation Models

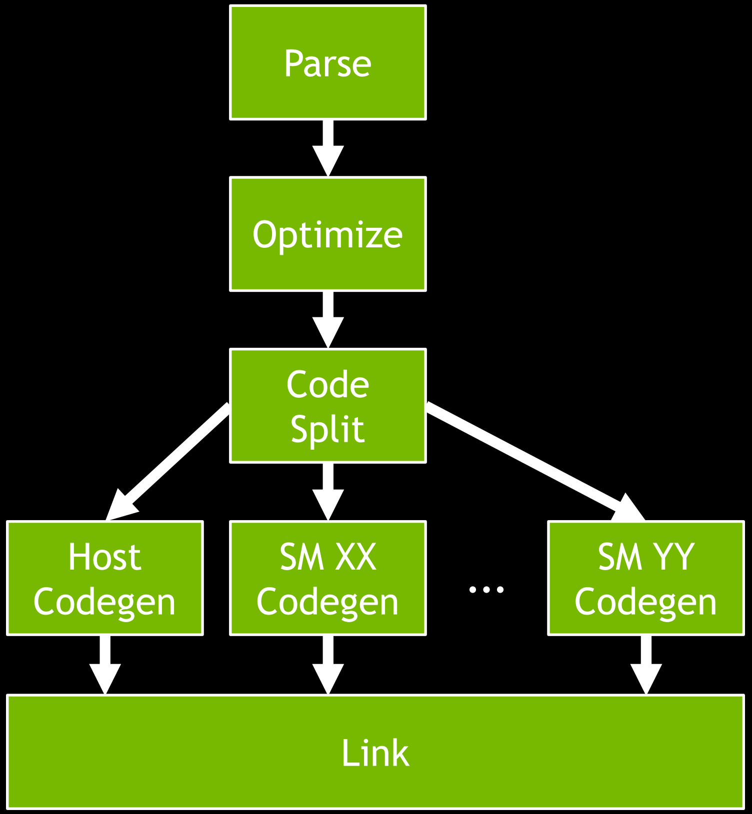
Split (NVCC)



Heterogeneous Compilation Models



Why Unified Heterogeneous Compilation?



Greater efficiency: parse, analyze, and optimize once instead of multiple times.

A single canonical representation of code provides greater visibility, analysis, interoperability, and control across host and device code.

- Increased robustness.
- Better diagnostics.
- Better language support.
- More heterogeneous features.

libnv++

Coming soon in NVC++!

libnv++

The C++ Standard Library for All Programming Models

ISO Standard Parallelism

CUDA

OpenACC

OpenMP™

Host Compiler’s Standard Library (GCC, MSVC, etc)

#include <…> std::	ISO C++, __host__ only. Complete, strictly conforming to Standard C++.

#include <cuda/std/…> cuda::std::	CUDA C++, __host__ __device__. Subset, strictly conforming to Standard C++.
#include <cuda/…> cuda::	CUDA C++, __host__ __device__. Conforming extensions to Standard C++.

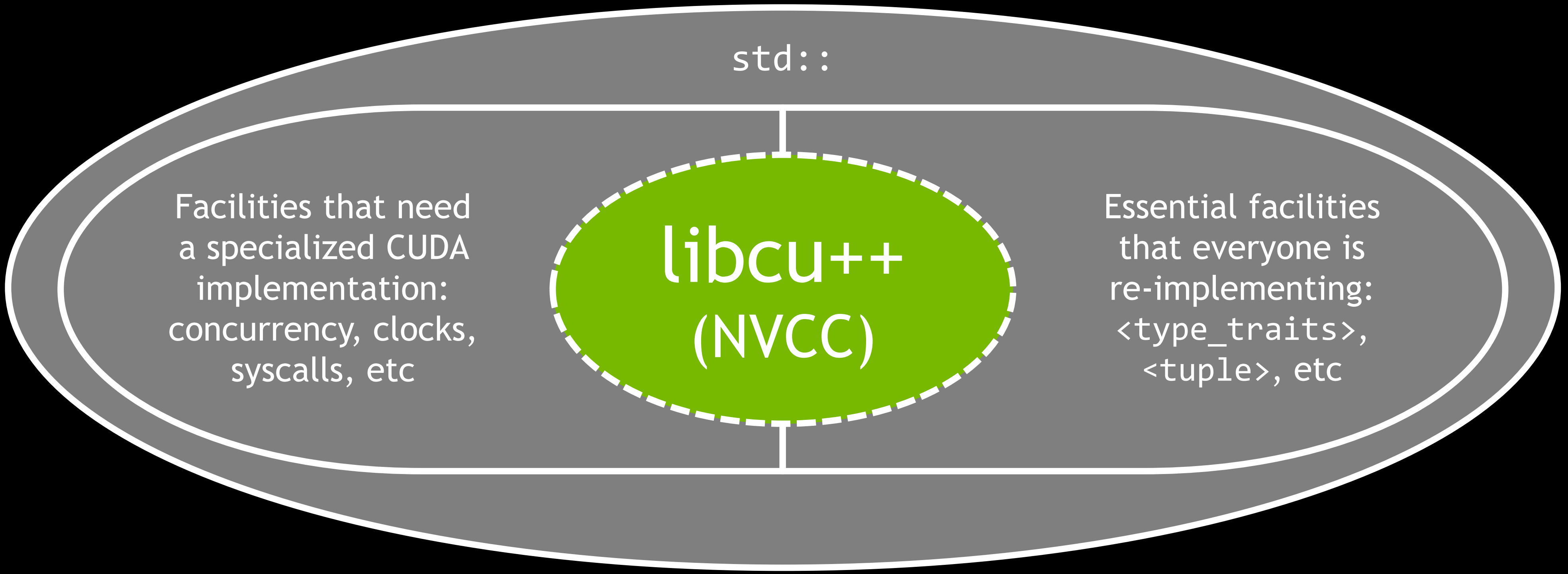
libcu++ (NVCC)

libcu++ does not interfere with or replace your host Standard Library.

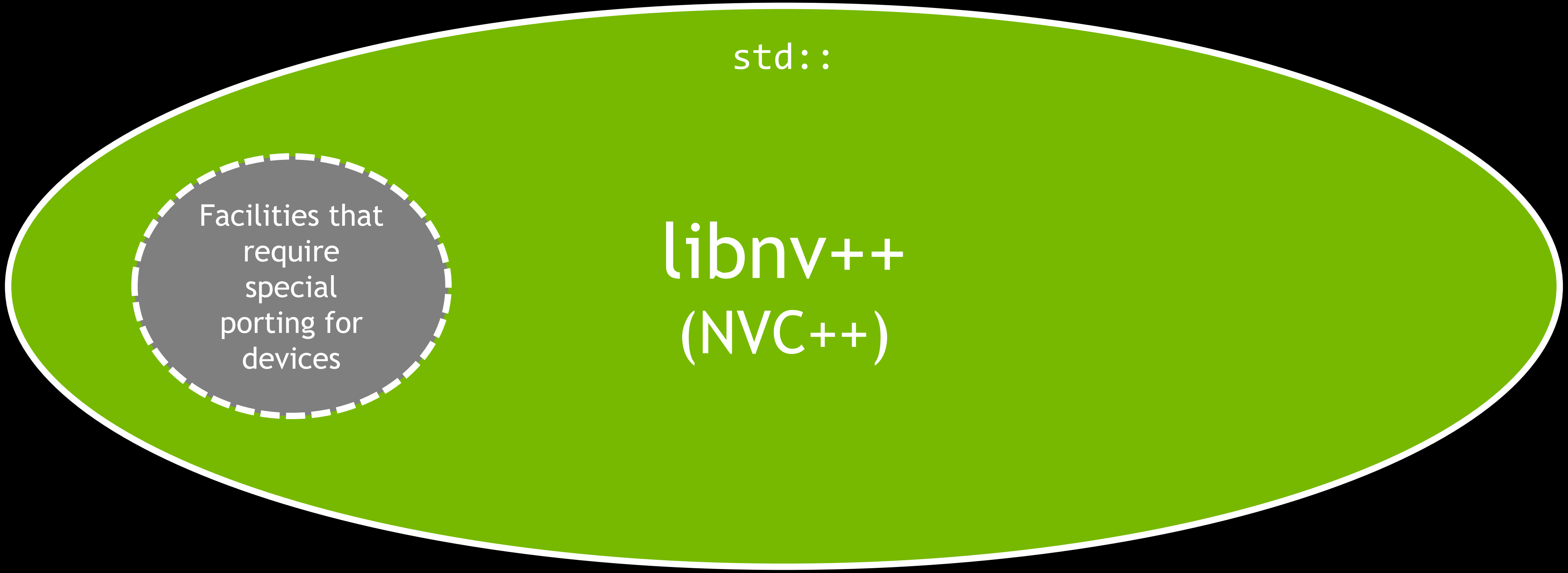
`#include <…>` `std::`	ISO C++, `__host__` only. Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>` `cuda::std::`	CUDA C++, `__host__ __device__`. Subset, strictly conforming to Standard C++.
`#include <cuda/…>` `cuda::`	CUDA C++, `__host__ __device__`. Conforming extensions to Standard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

libnv++ (NVC++)

libnv++ is your Standard Library. There is no host Standard Library.



libcu++ is not a complete Standard Library today.
Each release adds more.



libnv++ is a complete C++ Standard Library.
Everything works on host; most things work on device.

std::

Facilities that require special porting for devices

libnv++
(NVC++)

It's easier for us to deliver heterogeneous Standard Library features in NVC++:

- We control the entire toolchain.
- We only have to support one compiler frontend.
- NVC++ execution space inference means we don't have to add `__host__` and `__device__` everywhere.

libnv++ is a complete C++ Standard Library.
Everything works on host; most things work on device.

Learn More: [Inside NVC++ and NVFORTRAN, Bryce Adelstein Lelbach [S31358]]

	std::		cuda::[std::]	
	Host	Device	Host	Device
GCC, MSVC, etc	☑	☒	☒	☒

	`std::`		`cuda::[std::]`	
	Host	Device	Host	Device
GCC, MSVC, etc	☑	☒	☒	☒
libcu++ (NVCC)	☒	☒	☑	☑

	`std::`		`cuda::[std::]`	
	Host	Device	Host	Device
GCC, MSVC, etc	☑	☒	☒	☒
libcu++ (NVCC)	☒	☒	☑	☑
libnv++ (NVC++)	☑	☑	☑	☑

Two Standard Library Choices for NVC++

Coming soon!

`nvc++ -stdlib=libstdc++`

`std::` is host only.
ABI compatible with GCC.

`nvc++ -stdlib=libnv++`

`std::` is heterogeneous.
Not ABI compatible with GCC.

libnv++

Coming soon in NVC++!

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)
`atomic<T>` (SM60+) Type Traits

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)
atomic<T> (SM60+) Type Traits

1.1.0 (CUDA 11.0)
atomic<T>::wait/notify (SM70+) barrier (SM70+) latch (SM70+) *_semaphore (SM70+) cuda::memcpy_async (SM70+) chrono:: Clocks & Durations ratio<Num, Denom>

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

`atomic<T>` (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

`atomic<T>::wait/notify` (SM70+)
`barrier` (SM70+)
`latch` (SM70+)
`*_semaphore` (SM70+)
`cuda::memcpy_async` (SM70+)
`chrono::` Clocks & Durations
`ratio<Num, Denom>`

1.2.0 (CUDA 11.1)

`cuda::pipeline (SM80+)`

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

```
namespace cuda {

enum thread_scope {
  thread_scope_system,
  thread_scope_device,
  thread_scope_block,
  thread_scope_thread
};

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

} // namespace cuda
```

```
namespace cuda {

enum thread_scope {
  thread_scope_system,
  thread_scope_device,
  thread_scope_block,
  thread_scope_thread
};

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

} // namespace cuda
```

```
namespace cuda {

enum thread_scope {
  thread_scope_system,
  thread_scope_device,
  thread_scope_block,
  thread_scope_thread
};

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

} // namespace cuda
```

```
namespace cuda {

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

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

} // namespace cuda
```

```
namespace cuda {

enum thread_scope {
  thread_scope_system, // All threads.
  thread_scope_device, // All threads on the same processor.
  thread_scope_block,
  thread_scope_thread
};

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

} // namespace cuda
```

```
namespace cuda {

enum thread_scope {
  thread_scope_system, // All threads.
  thread_scope_device, // All threads on the same processor.
  thread_scope_block,  // All threads in a block.
  thread_scope_thread
};

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

} // namespace cuda
```

```
namespace cuda {

enum thread_scope {
  thread_scope_system, // All threads.
  thread_scope_device, // All threads on the same processor.
  thread_scope_block,  // All threads in a block.
  thread_scope_thread  // A single thread.
};

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

} // namespace cuda
```

```
// In NVCC and NVC++:
namespace cuda::std {

template <typename T>
using atomic = cuda::atomic<T, cuda::thread_scope_system>;

} // namespace cuda::std


// In NVC++ only:
namespace std {

template <typename T>
using atomic = cuda::atomic<T, cuda::thread_scope_system>;

} // namespace std
```

```
__host__ __device__ void
write_then_signal_flag(volatile int& flag, int& data, int v) {
  // ^^^ volatile was "notionally right" in legacy CUDA C++.
  data = v;

  // vvv "Works" on some platforms but is UB (volatile != atomic).
  flag = 1;
}
```

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

```
__host__ __device__ void
write_then_signal_flag(int& flag, int& data, int v) {

  data = v;
  __threadfence_system(); // <- Should be fused on the operation.

  atomicExch(&flag, 1); // <- Ideally we want an atomic store.
}
```

```
__host__ __device__ void
write_then_signal_better(atomic<bool>& flag, int& data, int v) {
  data = v;
  flag = true;
}
```

```
__host__ __device__ void
write_then_signal_even_better(atomic<bool>& flag, int& data, int v) {
  data = v;
  flag.store(true, memory_order_release);
}
```

```
__host__ __device__ void
write_then_signal_excellent(atomic<bool>& flag, int& data, int v) {
  data = v;
  flag.store(true, memory_order_release);
  flag.notify_all();
}
```

```
__host__ __device__ int
poll_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.

  return data;
}
```

```
__host__ __device__ int
poll_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_then_read(int& flag, int& data) {

  while (1 != atomicAdd(flag, 0)) // <- Should be an 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_then_read_better(atomic<bool>& flag, int& data) {
  while (!flag)
    ; // <- Spinloop without backoff is bad under contention.
  return data;
}
```

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

```
__host__ __device__ void increment(int& a) {
  #ifdef __CUDA_ARCH__
    atomicAdd(&a, 1);
  #else
    // What do I write here for all the CPUs & compilers I support?
  #endif
}
```

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

```
atomic<bool> sflag;
write_then_signal(sflag, ...); // Ok;
                               // expects and got system atomic type.

atomic<bool, thread_scope_device> d;
write_then_signal(dflag, ...); // Compile error;
                               // expects system atomic type.
```

```
// Called by many threads with different inputs and the same bins.
__host__ __device__ void
letter_histogram(string_view in, array<uint64_t, 256>& bins) {
  for (auto c : in)
    atomicAdd(bins[c], 1);
}
```

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

2.0.0

atomic_ref<T> (SM60+)

`std::atomic<T>` holds a T.

```
template <struct T>
struct atomic {
private:
  T data; // exposition only
public:

  // ...
};
```

`std::atomic_ref<T>` doesn't hold a T; it holds a reference to a T.

```
template <struct T>
struct atomic_ref {
private:
  T& ptr; // Exposition only.
public:
  explicit atomic_ref(T&);

  // Same API as std::atomic.
};
```

Coming soon for C++11 and up in the NVIDIA C++ Standard Library.

```
// Called by many threads with different inputs and the same bins.
__host__ __device__ void
letter_histogram_better(string_view in, array<uint64_t, 256>& bins) {
  for (auto c : in)
    atomic_ref(bins[c]).fetch_add(1, memory_order_relaxed);
}
```

Coming soon for C++11 and up in the NVIDIA C++ Standard Library.

```
namespace cuda {

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

} // namespace cuda
```

Coming soon for C++11 and up in the NVIDIA C++ Standard Library.

```
// In NVCC and NVC++:
namespace cuda::std {

template <typename T>
using atomic_ref = cuda::atomic_ref<T, cuda::thread_scope_system>;

} // namespace cuda::std


// In NVC++ only:
namespace std {

template <typename T>
using atomic_ref = cuda::atomic_ref<T, cuda::thread_scope_system>;

} // namespace std
```

Coming soon for C++11 and up in the NVIDIA C++ Standard Library.

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

No direct sequential consistency & acquire/release semantics.

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` != atomicity.

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

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

2.0.0

atomic_ref<T> (SM60+)

```
namespace cuda {

template <thread_scope S = thread_scope_system,
          typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};

} // namespace cuda
```

```
namespace cuda {

template <thread_scope S = thread_scope_system,
          typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};

} // namespace cuda
```

```
namespace cuda {

template <thread_scope S = thread_scope_system,
          typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};

} // namespace cuda
```

```
namespace cuda {

template <thread_scope S = thread_scope_system,
          typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};

} // namespace cuda
```

```
__device__ void sync_block() {
  __syncthreads();
  // Only works in device code and only within blocks.
  // Synchronizes all threads in the block.

  // What if I need multiple barriers?
  // What if I only want some threads to participate?
  // What if I want to synchronize across blocks?
  // What if I want to synchronize with the host?
}
```

```
__host__ __device__ void
better_sync_block(barrier<thread_scope_block>& b) {
  b.arrive_and_wait();
}

__host__ __device__ void
better_sync(barrier<thread_scope_system>& b) {
  b.arrive_and_wait();
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  // ...
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  // ...
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  // ...
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  // ...
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.

    // ...
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.




    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    // ...
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.




    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    __syncthreads(); // Wait until everyone has read their neighbor points.
    // ...
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.




    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    __syncthreads(); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    // ...
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.




    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    __syncthreads(); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    __syncthreads(); // Make sure everyone sees the new boundary points.
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.




    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    __syncthreads(); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    __syncthreads(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve(double* local, double* global, size_t i, size_t j, size_t steps) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.





    __syncthreads(); // Wait until everyone has read their neighbor points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    __syncthreads(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_better(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.


    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_better(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.


    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_better(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.


    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_better(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left = global[i - 1];  // Load from slow memory.
    double right = global[j + 1]; // Load from slow memory.


    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_excellent(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left, right;
    memcpy_async(&left, global + (i - 1), sizeof(double), b);
    memcpy_async(&right, global + (j + 1), sizeof(double), b);

    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    global[i] = local[0]; // Store to slow memory.

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    global[j] = local[j - i]; // Store to slow memory.

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_excellent(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left, right;
    memcpy_async(&left, global + (i - 1), sizeof(double), b);
    memcpy_async(&right, global + (j + 1), sizeof(double), b);

    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    memcpy_async(global + i, local, sizeof(double), b);

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    memcpy_async(global + j, local + (j - i), sizeof(double), b);

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

```
__host__ __device__ void
solve_excellent(double* local, double* global, size_t i, size_t j, size_t steps, barrier<>& b) {
  for (size_t t = 0; t < steps; ++t) {
    // Get points from neighbors.
    double left, right;
    memcpy_async(&left, global + (i - 1), sizeof(double), b);
    memcpy_async(&right, global + (j + 1), sizeof(double), b);

    auto tok = b.arrive(); // Signal that we've read our neighbor's points.

    // Solve interior region.
    for (size_t x = 0; x < ((j - i) - 1); ++x)
      local[x] = stencil(local[x - 1], local[x], local[x + 1]);

    b.wait(move(tok)); // Wait until everyone has read their neighbor points.

    // Solve boundary points and make them visible to neighbors.
    local[0] = stencil(left, local[0], local[1]);
    memcpy_async(global + i, local, sizeof(double), b);

    local[j - i] = stencil(local[(j - i) - 1], local[j - i], right);
    memcpy_async(global + j, local + (j - i), sizeof(double), b);

    b.arrive_and_wait(); // Make sure everyone sees the new neighbor points.
  }
}
```

Learn More: Async GPU Programming in CUDA C++, Vishal Mehta and Gonzalo Brito [E31888]

Volta+ NVIDIA GPUs deliver and libcu++ exposes:

C++ Parallel Forward Progress Guarantees.

The C++ Memory Model.

Why does this matter?

Learn More: Designing (New) C++ Hardware, Olivier Giroux [CppCon 2017]

Volta+ NVIDIA GPUs deliver and libcu++ exposes:

C++ Parallel Forward Progress Guarantees.

The C++ Memory Model.

Why does this matter?

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

	Progress doesn’t depend on scheduler	Progress depends on scheduler	
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

	Progress doesn’t depend on scheduler	Progress depends on scheduler	
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

	Progress doesn't depend on scheduler	Progress depends on scheduler	
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

Parallel Forward Progress

Only Volta+

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 modern NVIDIA 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_;
};
```

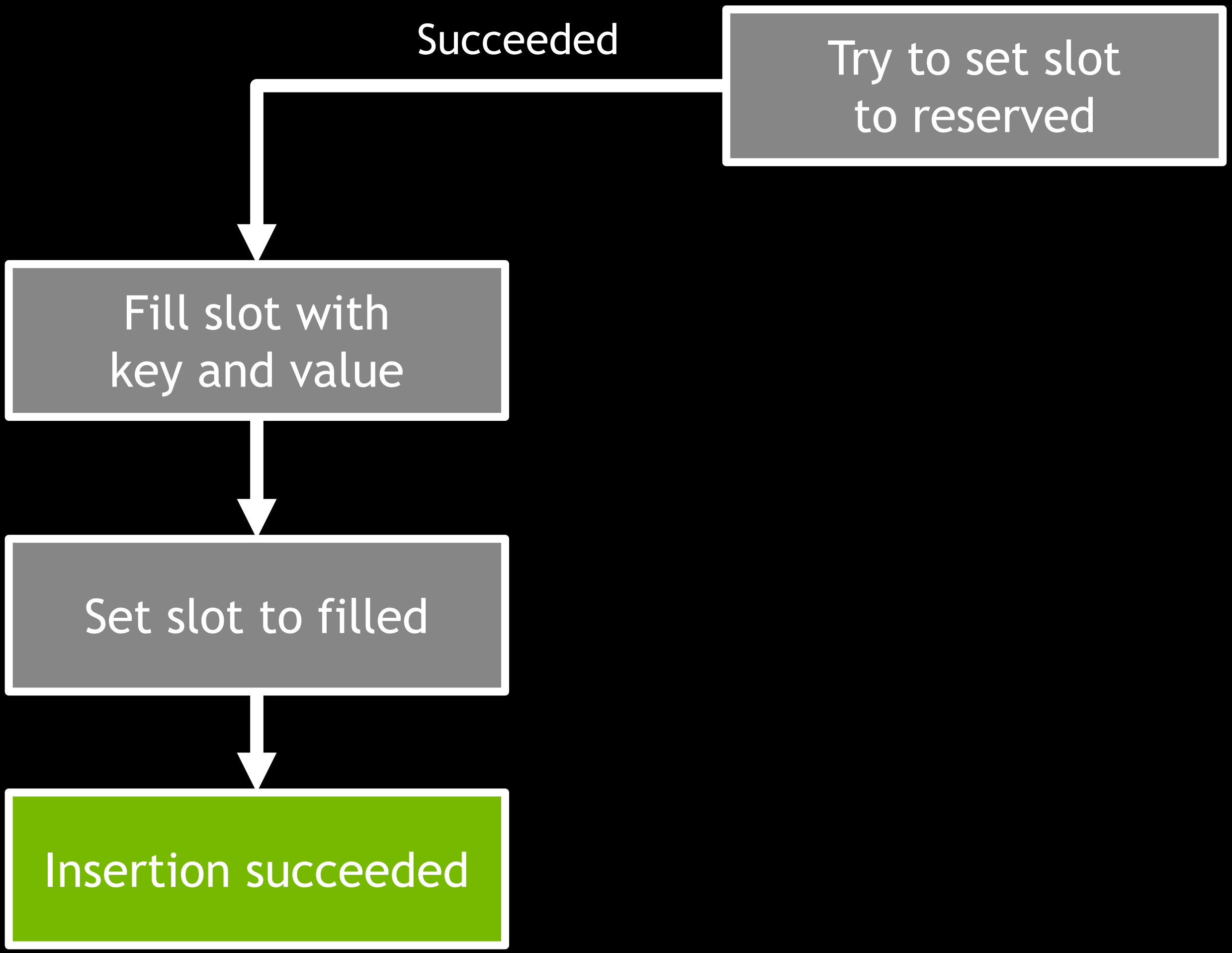
```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __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_;
};
```

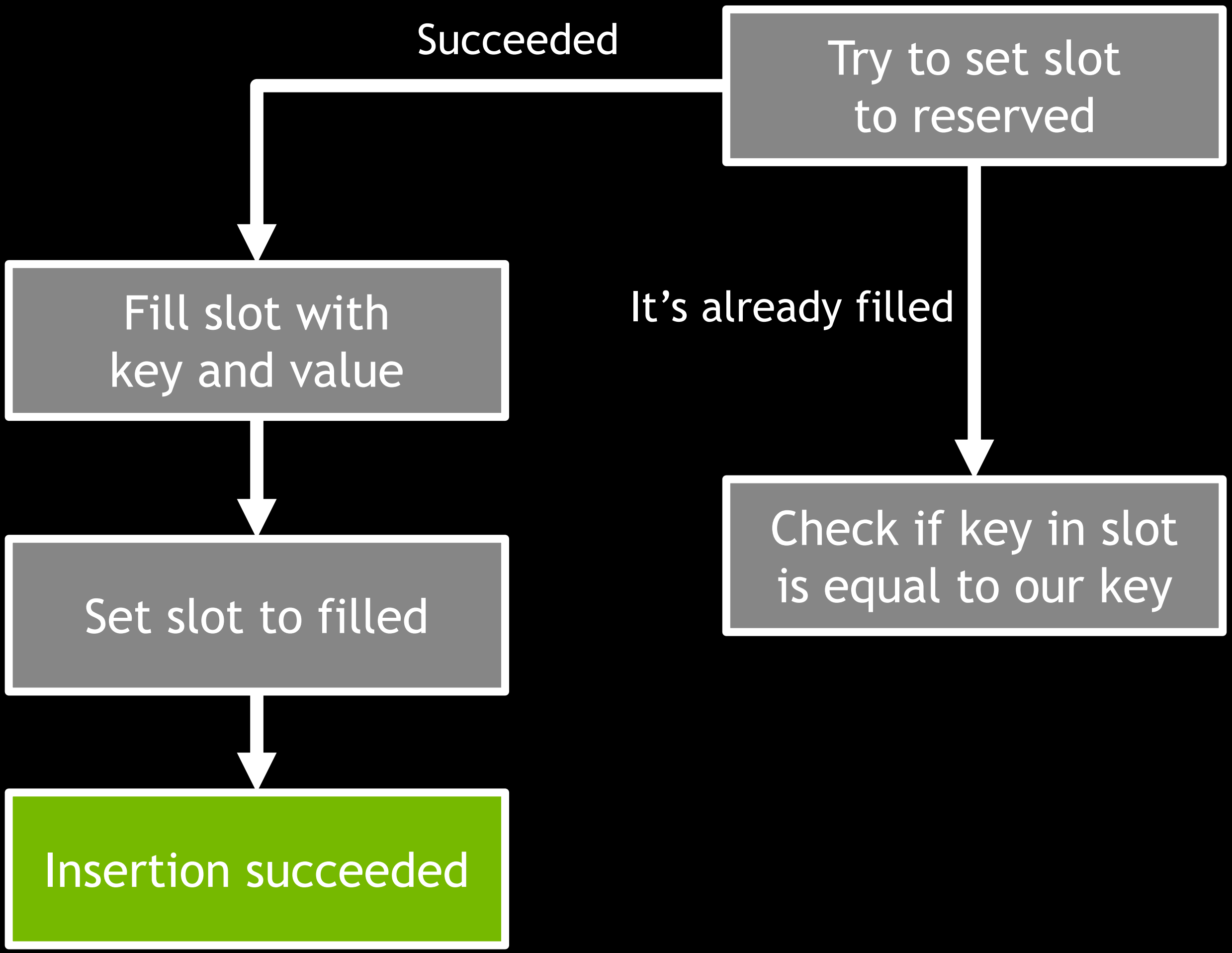
```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __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_;
};
```

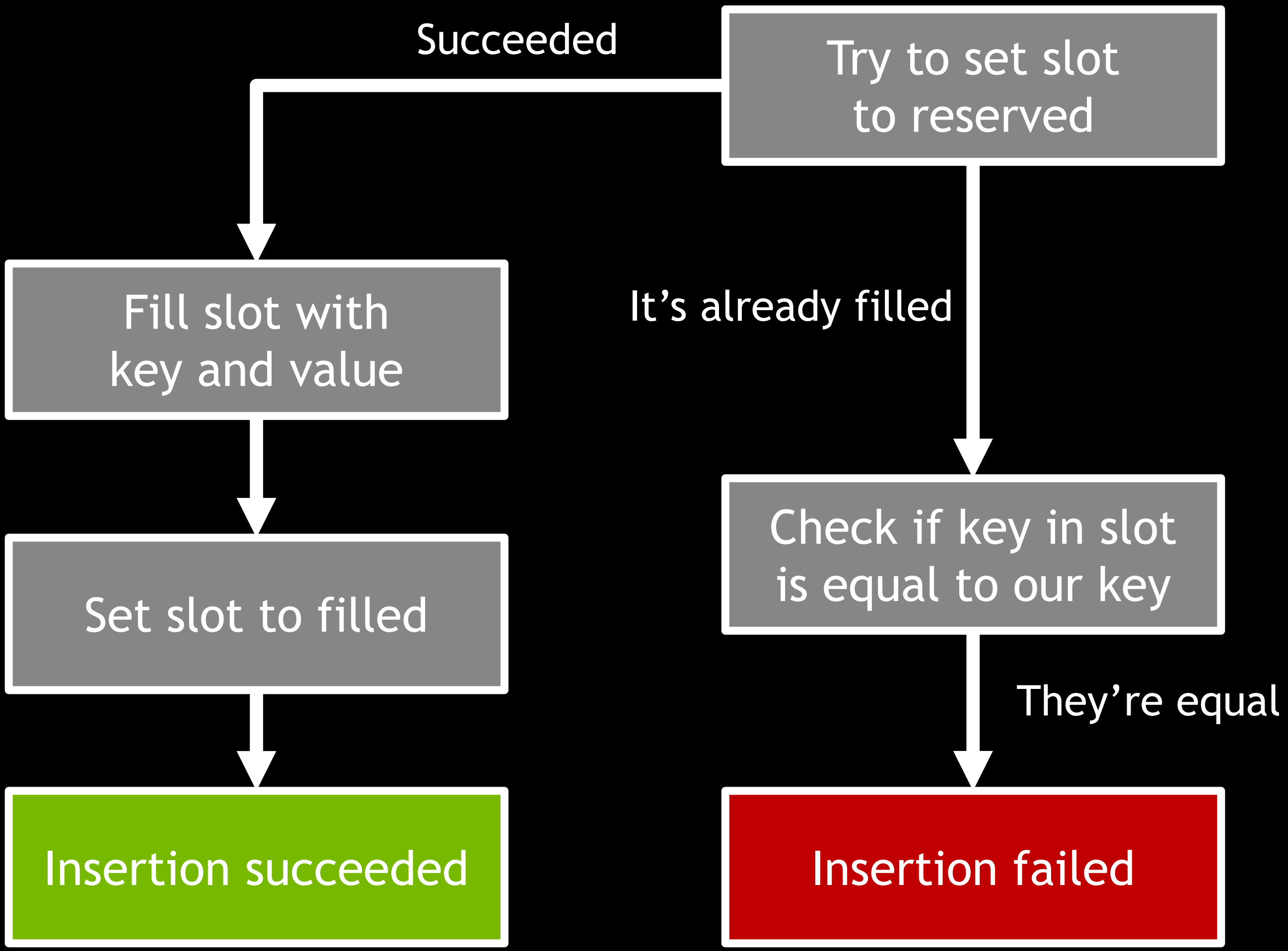
```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __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_;
};
```

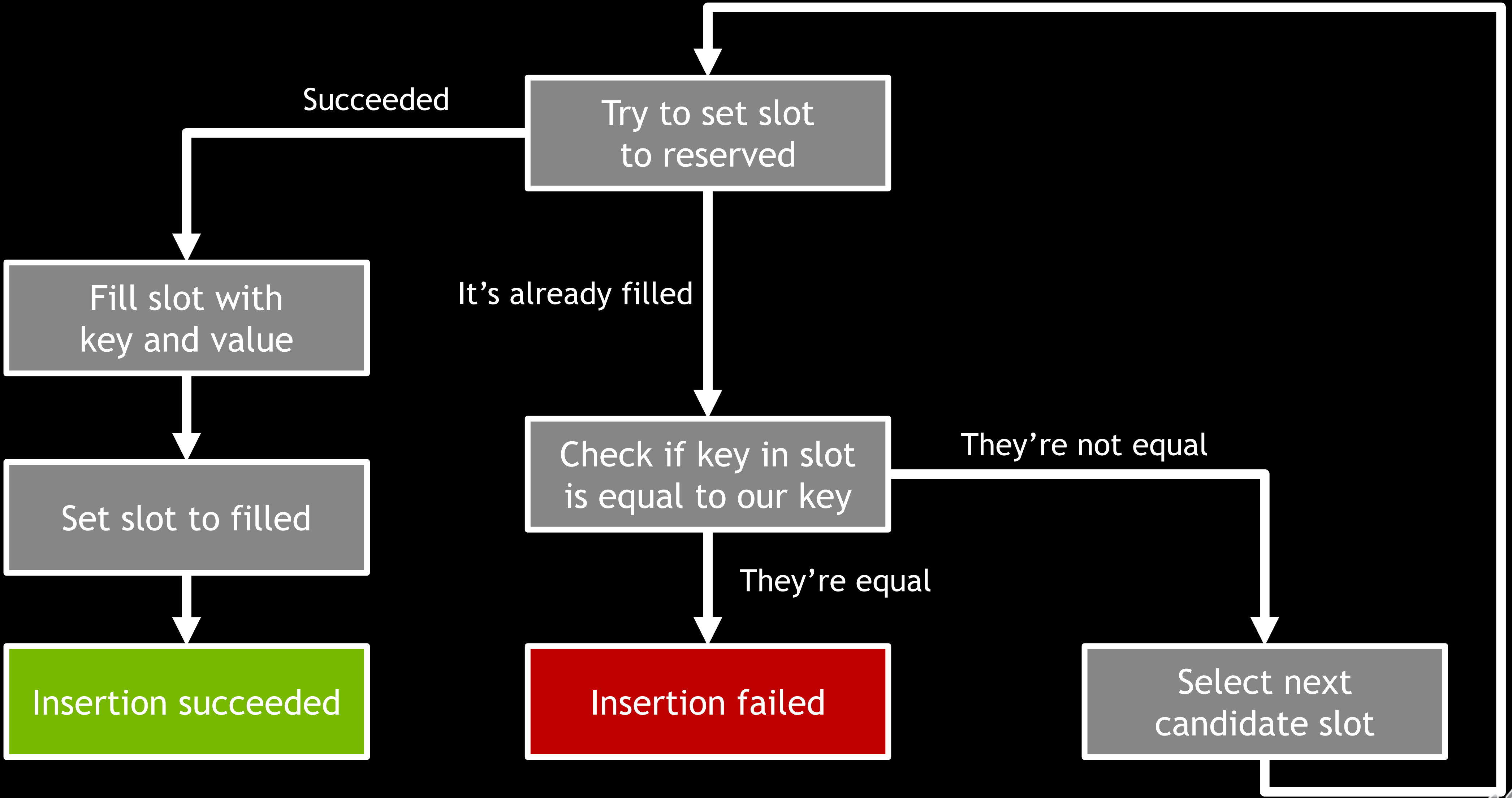
```
template <typename Key, typename Value,
          typename Hash  = hash<Key>,
          typename Equal = equal_to<Key>>
struct concurrent_insert_only_map {
  enum state_type { state_empty, state_reserved, state_filled };

  // ...

  __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_;
};
```

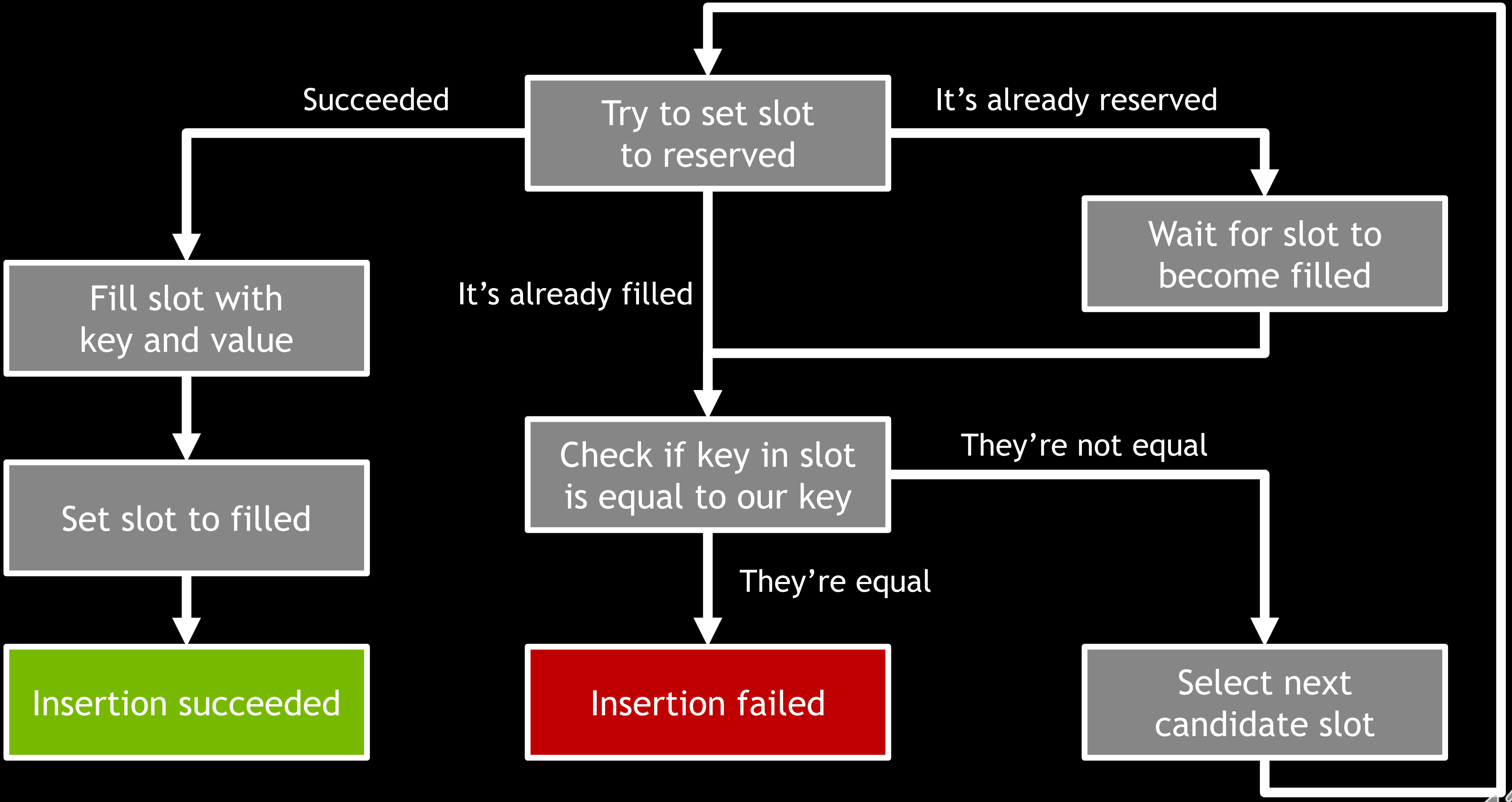
Try to set slot
to reserved


Succeeded
Try to set slot to reserved
Fill slot with key and value
Set slot to filled
Insertion succeeded


Succeeded
Try to set slot to reserved
Fill slot with key and value
It's already filled
Check if key in slot is equal to our key
Set slot to filled
Insertion succeeded


Succeeded
Try to set slot to reserved
Fill slot with key and value
It's already filled
Set slot to filled
Check if key in slot is equal to our key
They're equal
Insertion succeeded
Insertion failed


Try to set slot to reserved
Succeeded
Fill slot with key and value
Set slot to filled
Insertion succeeded
It’s already filled
Check if key in slot is equal to our key
They’re equal
Insertion failed
They’re not equal
Select next candidate slot



Try to set slot to reserved
Succeeded
It’s already reserved
Wait for slot to become filled
It’s already filled
Fill slot with key and value
Set slot to filled
Insertion succeeded
Check if key in slot is equal to our key
They’re not equal
They’re equal
Insertion failed
Select next candidate slot

```
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.
        // ...
      }
      // ...
    }
    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.
  }
};
```

```
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 the NVIDIA C++ Standard Library.

There's a whole new world of algorithms and data structures that can be accelerated with Volta+ NVIDIA GPUs using the NVIDIA C++ Standard Library.

Learn More: https://github.com/NVIDIA/cuCollections

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

`atomic<T>` (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

`atomic<T>::wait/notify` (SM70+)
`barrier` (SM70+)
`latch` (SM70+)
`*_semaphore` (SM70+)
`cuda::memcpy_async` (SM70+)
`chrono::` Clocks & Durations
`ratio<Num, Denom>`

1.2.0 (CUDA 11.1)

`cuda::pipeline` (SM80+)

1.3.0 (CUDA 11.2)

`tuple<T0, T1, ...>`

1.4.1 (CUDA 11.3)

`complex`
`byte`
`chrono::` Dates & Calendars

2.0.0

`atomic_ref<T>` (SM60+)
Memory Resources & Allocators

```cpp
namespace cuda {

enum class memory_kind {
  memory_kind_host     // Memory accessible only from host.
  memory_kind_device,  // Memory accessible only from device.
  memory_kind_unified, // Memory accessible from host & device.
  memory_kind_pinned,  // Memory accessible from host & device, but page-locked to host.
};

template <memory_kind Kind>
class memory_resource; // Type agnostic allocation. Owns state.

template <typename T, memory_kind Kind>
class allocator; // Typed allocation. Handle to a memory resource.

template <memory_kind Kind>
class stream_ordered_memory_resource; // Asynchronous allocation (cudaMallocAsync).

template <typename T, memory_kind Kind>
class stream_ordered_allocator; // Asynchronous allocation (cudaMallocAsync).

} // namespace cuda
```

```
namespace cuda {

enum class memory_kind {
  memory_kind_host      // Memory accessible only from host.
  memory_kind_device,   // Memory accessible only from device.
  memory_kind_unified,  // Memory accessible from host & device.
  memory_kind_pinned,   // Memory accessible from host & device, but page-locked to host.
};

template <memory_kind Kind>
class memory_resource; // Type agnostic allocation. Owns state.

template <typename T, memory_kind Kind>
class allocator; // Typed allocation. Handle to a memory resource.

template <memory_kind Kind>
class stream_ordered_memory_resource; // Asynchronous allocation (cudaMallocAsync).

template <typename T, memory_kind Kind>
class stream_ordered_allocator; // Asynchronous allocation (cudaMallocAsync).

} // namespace cuda
```

```cpp
namespace cuda {

enum class memory_kind {
  memory_kind_host      // Memory accessible only from host.
  memory_kind_device,   // Memory accessible only from device.
  memory_kind_unified,  // Memory accessible from host & device.
  memory_kind_pinned,   // Memory accessible from host & device, but page-locked to host.
};

template <memory_kind Kind>
class memory_resource; // Type agnostic allocation. Owns state.

template <typename T, memory_kind Kind>
class allocator; // Typed allocation. Handle to a memory resource.

template <memory_kind Kind>
class stream_ordered_memory_resource; // Asynchronous allocation (cudaMallocAsync).

template <typename T, memory_kind Kind>
class stream_ordered_allocator; // Asynchronous allocation (cudaMallocAsync).

} // namespace cuda
```

```
namespace cuda {

enum class memory_kind {
  memory_kind_host       // Memory accessible only from host.
  memory_kind_device,    // Memory accessible only from device.
  memory_kind_unified,   // Memory accessible from host & device.
  memory_kind_pinned,    // Memory accessible from host & device, but page-locked to host.
};

template <memory_kind Kind>
class memory_resource; // Type agnostic allocation. Owns state.

template <typename T, memory_kind Kind>
class allocator; // Typed allocation. Handle to a memory resource.

template <memory_kind Kind>
class stream_ordered_memory_resource; // Asynchronous allocation (cudaMallocAsync).

template <typename T, memory_kind Kind>
class stream_ordered_allocator; // Asynchronous allocation (cudaMallocAsync).

} // namespace cuda
```

```
namespace cuda {

enum class memory_kind {
  memory_kind_host     // Memory accessible only from host.
  memory_kind_device,  // Memory accessible only from device.
  memory_kind_unified, // Memory accessible from host & device.
  memory_kind_pinned,  // Memory accessible from host & device, but page-locked to host.
};

template <memory_kind Kind>
class memory_resource; // Type agnostic allocation. Owns state.

template <typename T, memory_kind Kind>
class allocator; // Typed allocation. Handle to a memory resource.

template <memory_kind Kind>
class stream_ordered_memory_resource; // Asynchronous allocation (cudaMallocAsync).

template <typename T, memory_kind Kind>
class stream_ordered_allocator; // Asynchronous allocation (cudaMallocAsync).

} // namespace cuda
```

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

`atomic<T>` (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

`atomic<T>::wait/notify` (SM70+)
`barrier` (SM70+)
`latch` (SM70+)
`*_semaphore` (SM70+)
`cuda::memcpy_async` (SM70+)
`chrono::` Clocks & Durations
`ratio<Num, Denom>`

1.2.0 (CUDA 11.1)

`cuda::pipeline` (SM80+)

1.3.0 (CUDA 11.2)

`tuple<T0, T1, ...>`

1.4.1 (CUDA 11.3)

`complex`
`byte`
`chrono::` Dates & Calendars

2.0.0

`atomic_ref<T>` (SM60+)
Memory Resources & Allocators
`cuda::stream_view`

```
namespace cuda {

// Non-owning wrapper of a cudaStream_t.
struct stream_view {
  constexpr stream_view(cudaStream_t stream) : __stream{stream} {}
  stream_view(int) = delete;              // Disallow construction from 0 or
  stream_view(std::nullptr_t) = delete; // nullptr to prevent ambiguity.
  constexpr cudaStream_t get() const noexcept { return __stream; }
private:
  cudaStream_t __stream{0};
};

} // namespace cuda
```

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

2.0.0

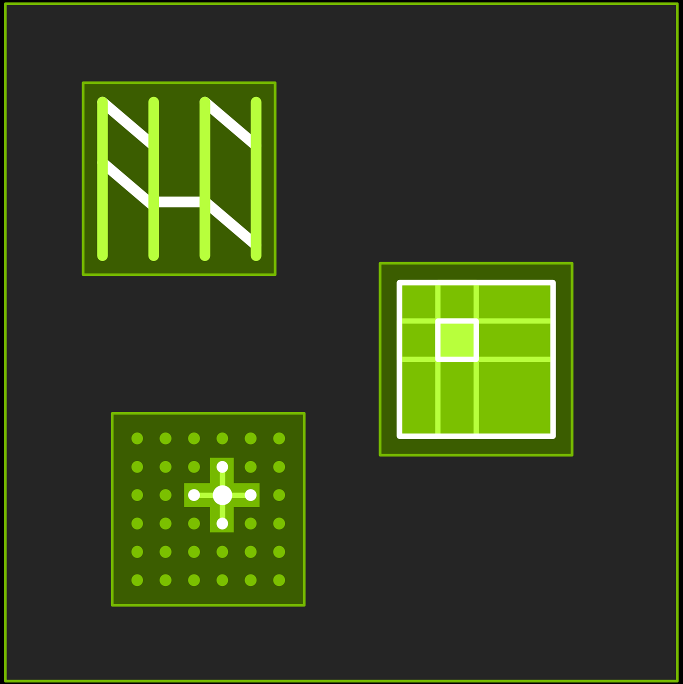
Memory Resources & Allocators
atomic_ref<T> (SM60+)
cuda::stream_view

Future

Executors
Range Factories & Adaptors
Parallel Range Algorithms
Parallel Linear Algebra Algorithms
mdspan<T, ...>
...

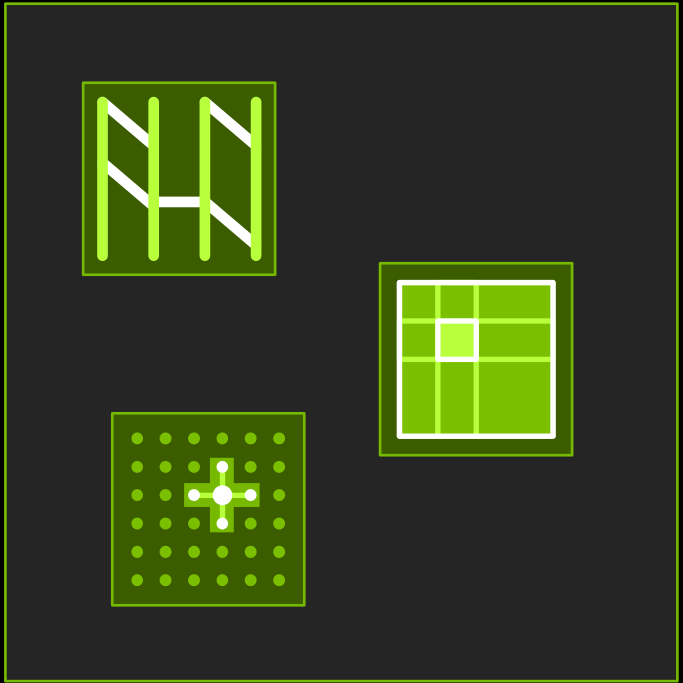
Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

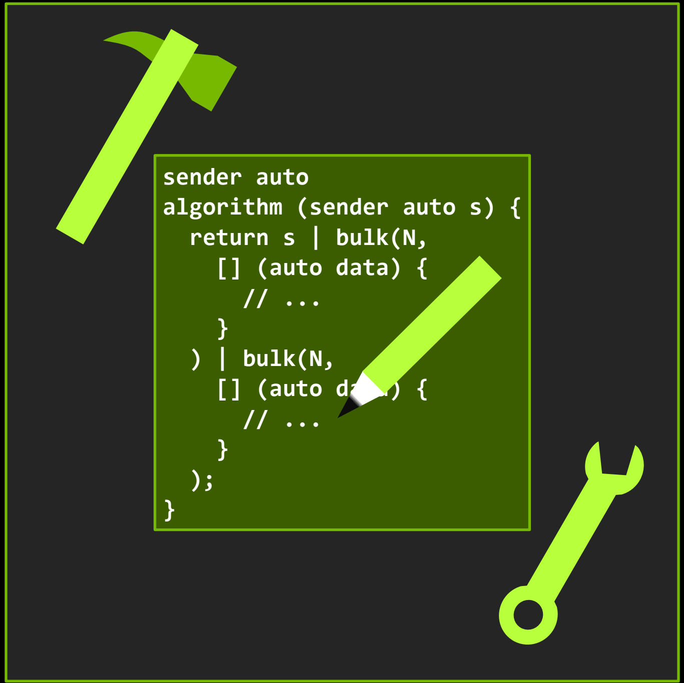


Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

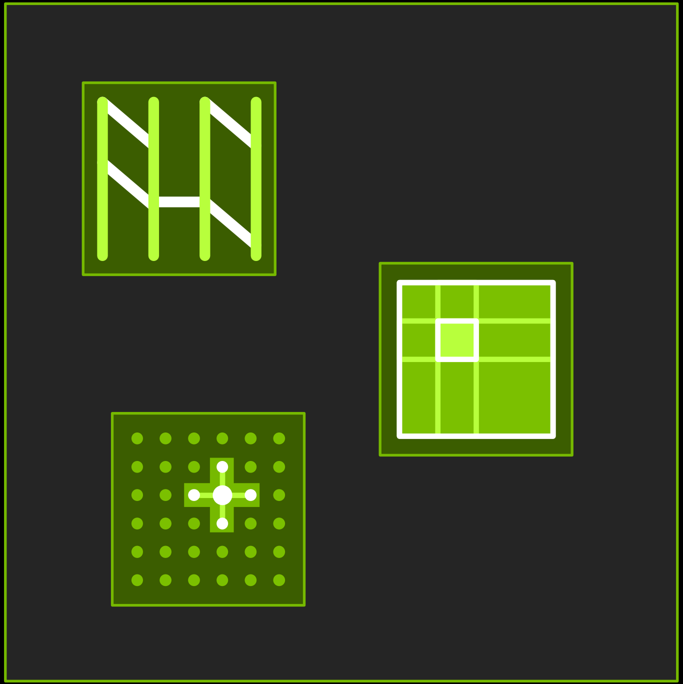


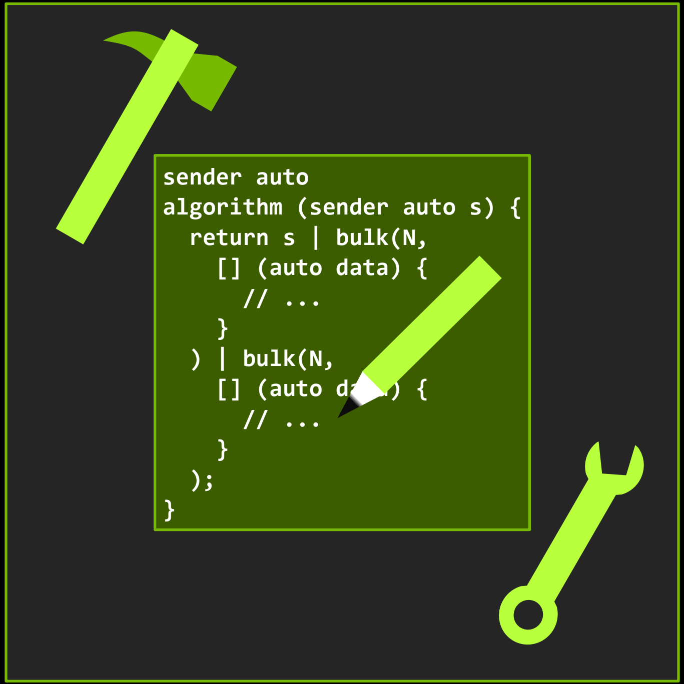
Tools to Write Your Own Parallel Algorithms that Run Anywhere

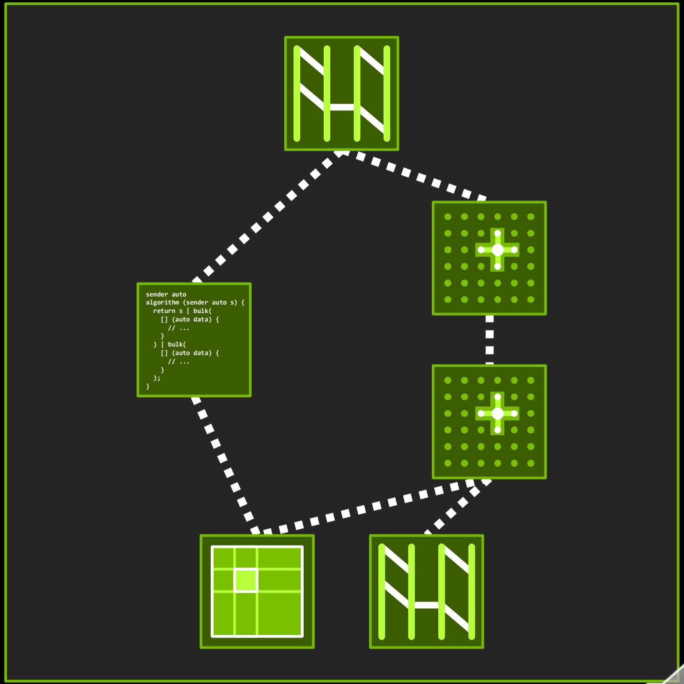


Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries	Tools to Write Your Own Parallel Algorithms that Run Anywhere	Mechanisms for Composing Parallel Invocations into Task Graphs

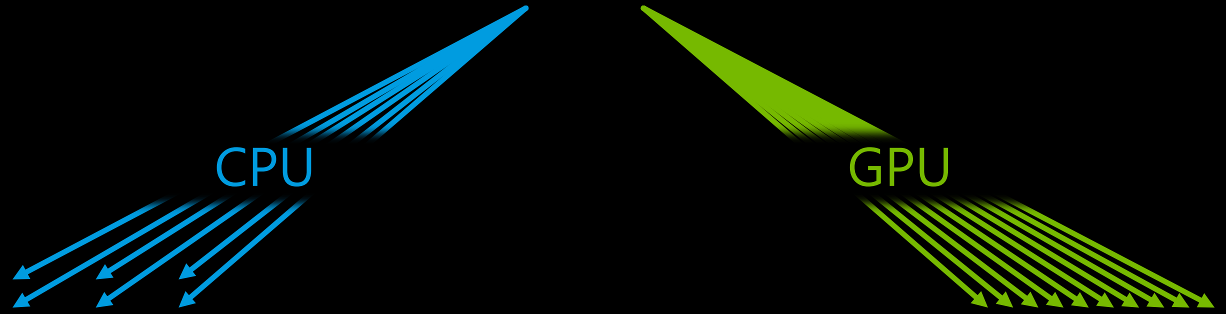






Standard Parallel Algorithms

```
std::vector<double> x(/* ... */), y(/* ... */);
double dot_product = std::transform_reduce(std::execution::par,
                                           x.begin(), x.end(), y.begin());
```



`nvc++ -stdpar=multicore`
NEW in 21.3!

`nvc++ -stdpar=gpu`
Since 20.7

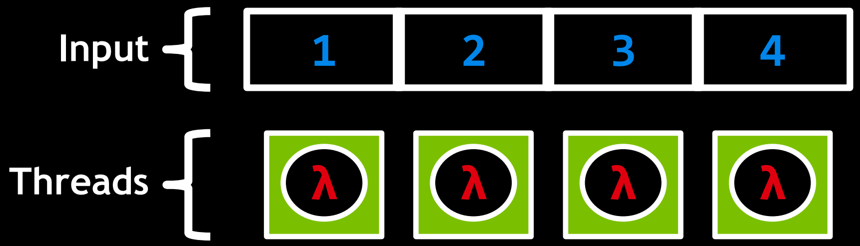
Learn More: [Inside NVC++ and NVFORTRAN, Bryce Adelstein Lelbach [S31358]]

Standard Parallel Algorithms & Ranges

Iterating Indices

CUDA C++

```
thrust::counting_iterator b(1);
thrust::counting_iterator e(N);
thrust::for_each(b, e,
  [] (auto idx) { /* ... */ });
```



On the NVIDIA C++ Standard Library roadmap!

Standard Parallel Algorithms & Ranges

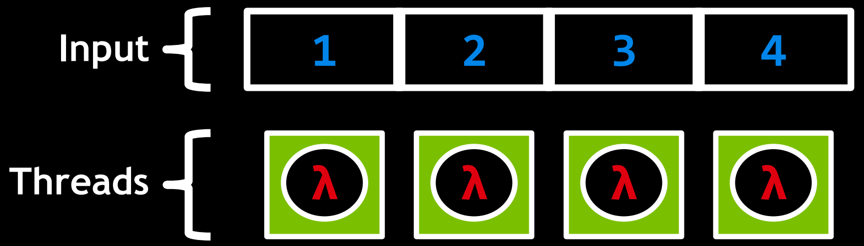
Iterating Indices

CUDA C++

```
thrust::counting_iterator b(1);
thrust::counting_iterator e(N);
thrust::for_each(b, e,
  [] (auto idx) { /* ... */ });
```

Standard C++20

```
auto v = std::views:iota(1, N);
std::for_each(
  std::execution::par,
  begin(v), end(v),
  [] (auto idx) { /* ... */ });
```



On the NVIDIA C++ Standard Library roadmap!

Standard Parallel Algorithms & Ranges

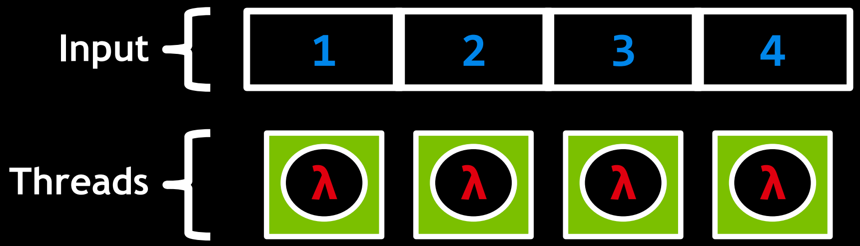
Iterating Indices

CUDA C++

```
thrust::counting_iterator b(1);
thrust::counting_iterator e(N);
thrust::for_each(b, e,
  [] (auto idx) { /* ... */ });
```

Standard C++ (With Parallel Ranges)

```
std::ranges::for_each(
  std::execution::par,
  std::views::iota(1, N),
  [] (auto idx) { /* ... */ });
```



On the C++2x and NVIDIA C++ Standard Library roadmap!

Standard Parallel Algorithms & Ranges

Iterating Multi-Dimensional Indices With C++23

```
std::span A(/*...*/);
std::span B(/*...*/);

auto v = std::views::cartesian_product(
  std::views::iota(0, N),
  std::views::iota(0, M));

std::for_each(std::execution::par,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[i * M + j] = A[j * N + i];
  });
```

On the C++23 and NVIDIA C++ Standard Library roadmap!

Standard Parallel Algorithms & Ranges

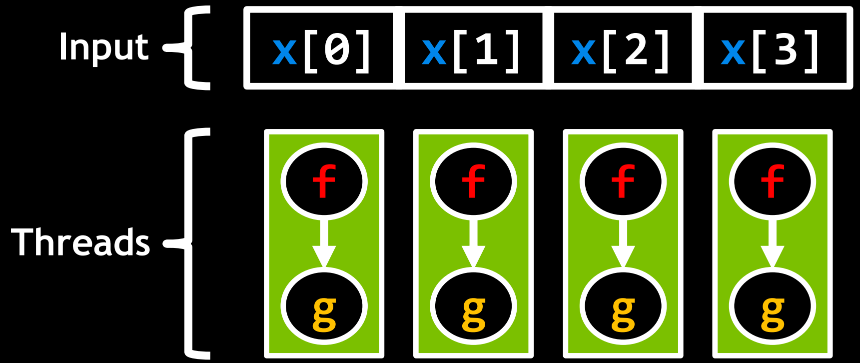
Composition and Asynchrony

CUDA C++

```
thrust::transform_iterator b(begin(x), f);
thrust::transform_iterator e(begin(x), f);
thrust::for_each(std::execution::par,
                 b, e, g);
```

Standard C++20

```
auto v = std::views::transform(x, f);
std::for_each(std::execution::par,
              begin(v), end(v), g);
```



On the NVIDIA C++ Standard Library roadmap!

Standard Parallel Algorithms & Ranges

Composition and Asynchrony

CUDA C++

```
template <typename U, typename C>
__global__ transform(U* u, size_t N, C c) {
  auto i = blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) u[i] = c(u[i]);
}

transform<<<((N-1)/128)+1, 128>>>(u, N, f);
transform<<<((N-1)/128)+1, 128>>>(u, N, g);
transform<<<((N-1)/128)+1, 128>>>(u, N, h);

cudaError_t err = cudaDeviceSynchronize();
assert(err == cudaSuccess);
```

Standard C++20

```
std::span u(/*...*/);

auto v = u
       | std::views::transform(f)
       | std::views::transform(g);

std::for_each(std::execution::par,
              begin(v), end(v), h);
```

On the NVIDIA C++ Standard Library roadmap!

Standard Linear Algebra

```
std::mdspan A(/* ... */, N, M);
std::mdspan x(/* ... */, M);
std::mdspan y(/* ... */, N);

// y = 3.0 * A * x + 2.0 * y
std::matrix_vector_product(
  std::execution::par,
  std::scaled_view(3.0, A), x,
  std::scaled_view(2.0, y), y);
```

A modern Standard C++ abstraction based on the BLAS.

Portable interface to both sequential and parallel linear algebra libraries on whatever platform you are on. On NVIDIA platforms, accelerated by cuBLAS, cuTENSOR and NVIDIA Tensor Cores.

`std::mdspan`: non-owning multi-dimensional span type.

On the C++2x and NVIDIA C++ Standard Library roadmap!

Standard Executors

```
std::span v(/* ... */);

std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] += v[idx - 1]; })
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and managing tasks.

They support both lazy and eager execution.

They provide a way to author kernels in Standard C++ that can run on any system.

On the C++2x and NVIDIA C++ Standard Library roadmap!

Standard Executors

```
std::span v(/* ... */);

std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] += v[idx - 1]; })
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors parameterize execution resources, allowing us to write code that can run anywhere.

Executors are a portable abstraction for composing and managing tasks.

They support both lazy and eager execution.

They provide a way to author kernels in Standard C++ that can run on any system.

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] += v[idx - 1]; })
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and managing tasks.

They support both lazy and eager execution.

They provide a way to author kernels in Standard C++ that can run on any system.

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] +=
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

nd

eager execution.

Sender Factories and Adaptors	
`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).

They provide a way to author kernels in Standard C++ that can run on any system.

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] +=
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Sender Factories and Adaptors	
`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).
`sender bulk(sender last, shape n, invocable f);`	Spawn `n` tasks that call `f` after `last` has completed (adaptor).

Executors are a portable abstraction for composing and

nd

uthor

kernels in Standard C++ that can run on any system.

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] +=
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

nd

uthor
that

Sender Factories and Adaptors

`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).

`sender bulk(sender last,`
`            shape n,`
`            invocable f);`
	Spawn `n` tasks that call `f` after `last` has completed (adaptor).

`sender just(T t);`	Pass the value `t` to the next work item (factory).

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
       | std::bulk(N,
           [](auto idx)
           { if (idx > 0) v[idx] +=
       | std::bulk(N,
           [](auto idx)
           { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

nd

uthor
that

Sender Factories and Adaptors	
`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).
`sender bulk(sender last, shape n, invocable f);`	Spawn `n` tasks that call `f` after `last` has completed (adaptor).
`sender just(T t);`	Pass the value `t` to the next work item (factory).
`sender then(sender last, invocable f);`	Call `f` with after `last` has completed (adaptor).

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] +=
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

nd

uthor
that

Sender Factories and Adaptors	
`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).
`sender bulk(sender last, shape n, invocable f);`	Spawn `n` tasks that call `f` after `last` has completed (adaptor).
`sender just(T t);`	Pass the value `t` to the next work item (factory).
`sender then(sender last, invocable f);`	Call `f` with after `last` has completed (adaptor).
`sender on(sender last, executor e);`	Switch to executor `e` for the next work item (adaptor).

Standard Executors

Senders represent asynchronous work that will be ready later (generalization of `std::future`).

Executors parameterize execution resources, allowing us to write code that can run anywhere.

```
std::sender auto s = std::schedule(cuda::executor)
        | std::bulk(N,
            [](auto idx)
            { if (idx > 0) v[idx] +=
        | std::bulk(N,
            [](auto idx)
            { v[idx] *= v[idx]; });
```

`operator|` creates a sender dependent on another sender.
`f | g | h == h(g(f()))`

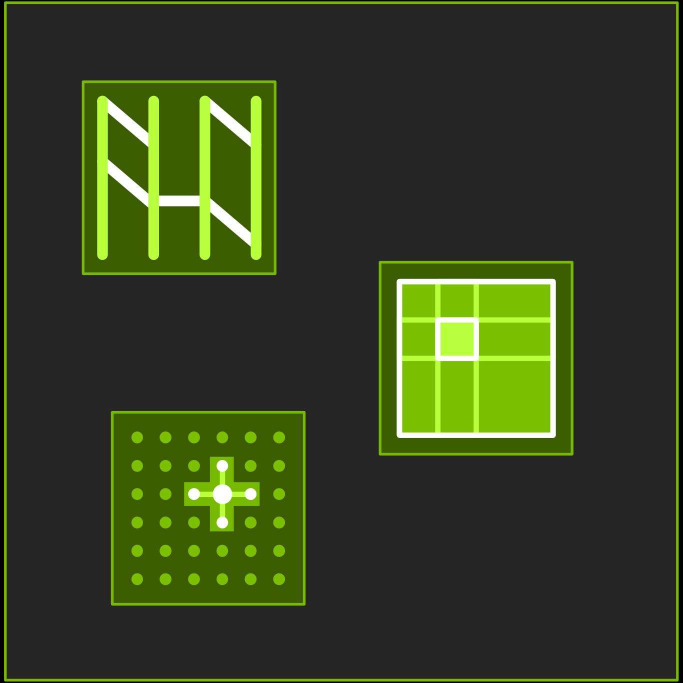
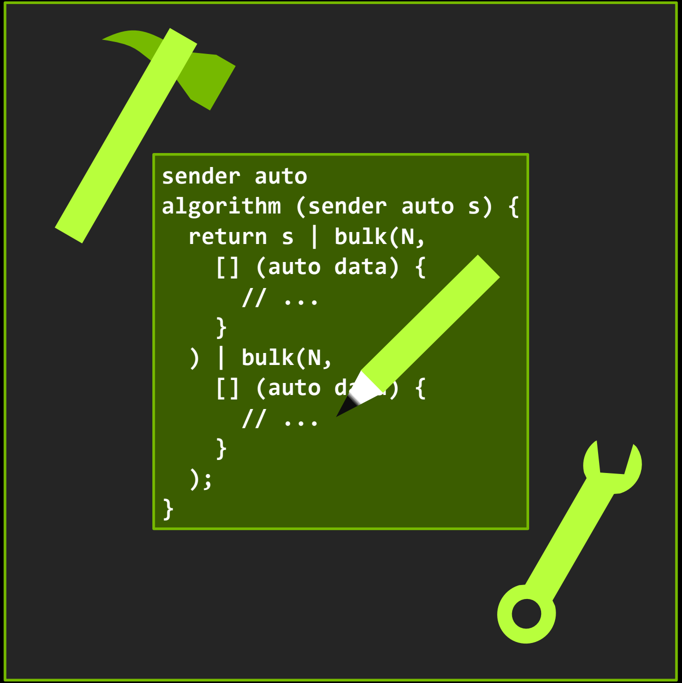
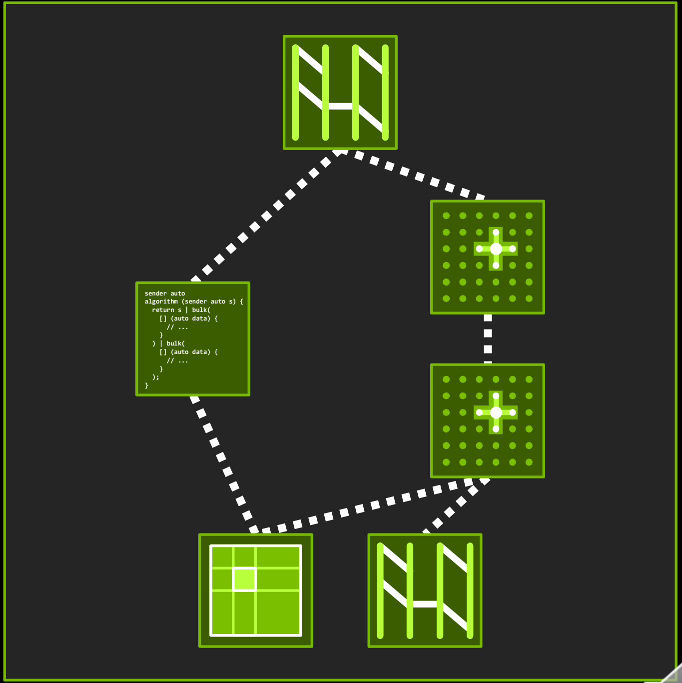
Executors are a portable abstraction for composing and

nd

uthor
that

Sender Factories and Adaptors	
`sender schedule(executor e);`	Start a chain of work on executor `e` (factory).
`sender bulk(sender last, shape n, invocable f);`	Spawn `n` tasks that call `f` after `last` has completed (adaptor).
`sender just(T t);`	Pass the value `t` to the next work item (factory).
`sender then(sender last, invocable f);`	Call `f` with after `last` has completed (adaptor).
`sender on(sender last, executor e);`	Switch to executor `e` for the next work item (adaptor).

Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries	Tools to Write Your Own Parallel Algorithms that Run Anywhere	Mechanisms for Composing Parallel Invocations into Task Graphs
		

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

atomic<T> (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

atomic<T>::wait/notify (SM70+)
barrier (SM70+)
latch (SM70+)
*_semaphore (SM70+)
cuda::memcpy_async (SM70+)
chrono:: Clocks & Durations
ratio<Num, Denom>

1.2.0 (CUDA 11.1)

cuda::pipeline (SM80+)

1.3.0 (CUDA 11.2)

tuple<T0, T1, ...>

1.4.1 (CUDA 11.3)

complex
byte
chrono:: Dates & Calendars

2.0.0

atomic_ref<T> (SM60+)
Memory Resources & Allocators
cuda::stream_view

Future

Executors
Range Factories & Adaptors
Parallel Range Algorithms
Parallel Linear Algebra Algorithms
mdspan<T, ...>
…

Learn More

Inside NVC++ and NVFORTRAN, Bryce Adelstein Lelbach [S31358]

A Deep Dive into the Latest HPC Software, Tim Costa [S31286]

CUDA: New Features and Beyond, Stephen Jones [S31857]

Thrust, CUB, and libcu++ User's Forum [CWES1801]

Thrust and the C++ Standard Algorithms, Conor Hoekstra [S31532]

Asynchronous GPU Programming in CUDA C++, Vishal Mehta and Gonzalo Brito [E31888]

Present and Future of Accelerated Computing Programming Approaches [S31146]

Future of Standard and CUDA C++ [CWES1802]

Controlling Data Movement to Boost Performance on NVIDIA Ampere [Blog Post]

Designing (New) C++ Hardware, Olivier Giroux [CppCon 2017]

High Radix Concurrent C++, Olivier Giroux [CppCon 2018]

https://github.com/NVIDIA/cuCollections

The C++ Standard Library for Your Entire System

https://github.com/NVIDIA/libcudacxx

@blelbach