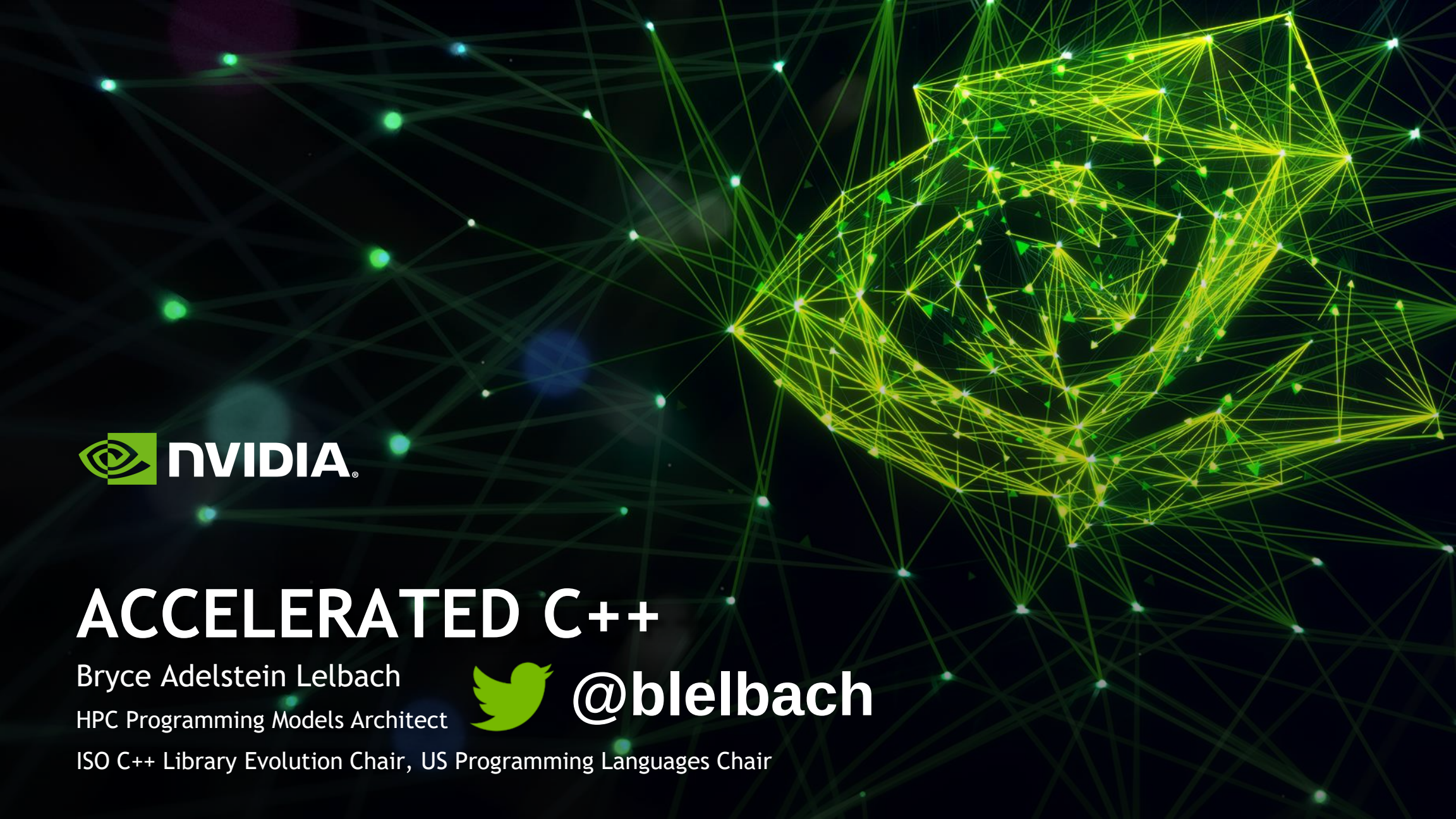


NVIDIA

ACCELERATED C++

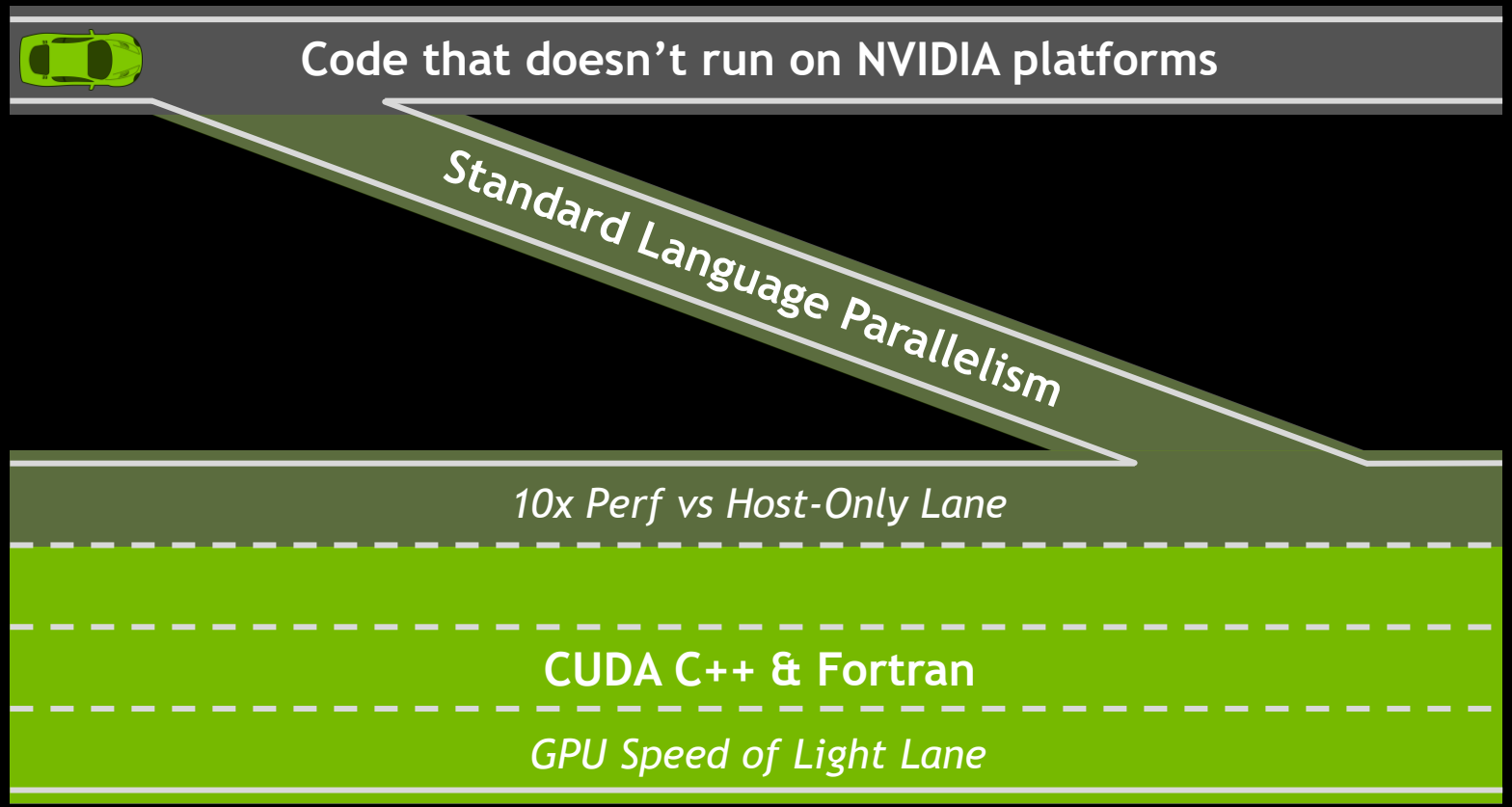
Bryce Adelstein Lelbach

@blelbach

HPC Programming Models Architect

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

Scientists Need On-Ramps



Accelerated Programming in 2020 and beyond

Libraries | Standard Languages | Directives | CUDA

```
std::transform(par, x, x+n, y, y,
  [=](float x, float y) {
    return y + a*x;
});
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

GPU Accelerated
Standard C++ and Fortran

```
#pragma acc data copy(x,y)
{

…

std::transform(par, x, x+n, y, y,
  [=](float x, float y) {
    return y + a*x;
});

…

}
```

Incremental Performance
Optimization with OpenACC

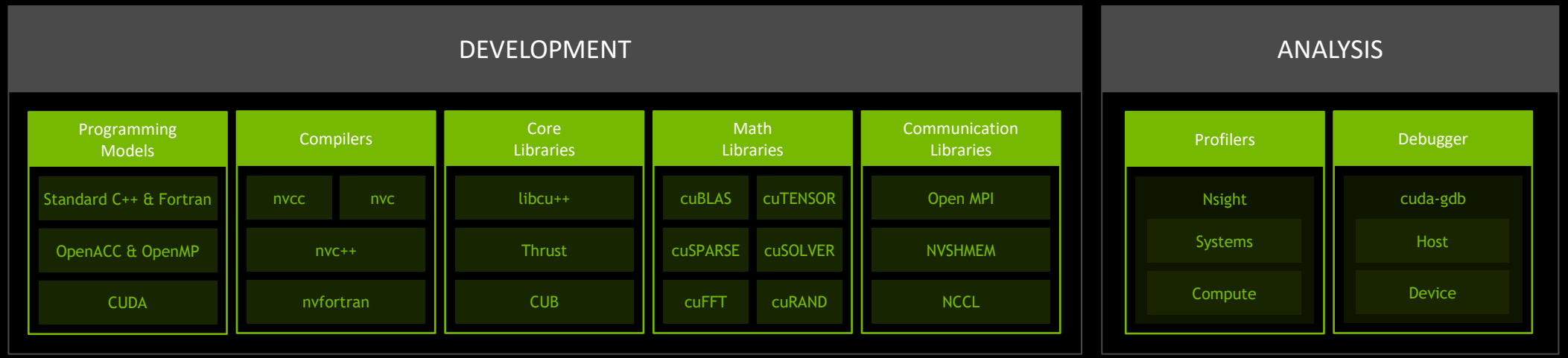
```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  …
  cudaMemcpy(d_x, x, …);
  cudaMemcpy(d_y, y, …);

  saxpy<<<(N+255)/256,256>>>(…);
  cudaMemcpy(y, d_y, …);
}
```

Maximize GPU Performance with
CUDA C++ and Fortran

GPU Accelerated Libraries

The NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, and in the Cloud



Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect

HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA

7-8 Releases Per Year | Freely Available

NVIDIA

```cpp
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                  Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro)
{
#if _OPENMP
  const Index_t threads = omp_get_max_threads();
  Index_t hydro_elem_per_thread[threads];
  Real_t dthydro_per_thread[threads];
#else
  Index_t threads = 1;
  Index_t hydro_elem_per_thread[1];
  Real_t dthydro_per_thread[1];
#endif
#pragma omp parallel firstprivate(length, dvovmax)
  {
    Real_t dthydro_tmp = dthydro ;
    Index_t hydro_elem = -1 ;
#if _OPENMP
    Index_t thread_num = omp_get_thread_num();
#else
    Index_t thread_num = 0;
#endif
#pragma omp for
    for (Index_t i = 0 ; i < length ; ++i) {
      Index_t indx = regElemlist[i] ;

      if (domain.vdov(indx) != Real_t(0.)) {
        Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx))+Real_t(1.e-20)) ;

        if ( dthydro_tmp > dtdvov ) {
          dthydro_tmp = dtdvov ;
          hydro_elem = indx ;
        }
      }
    }
    dthydro_per_thread[thread_num] = dthydro_tmp ;
    hydro_elem_per_thread[thread_num] = hydro_elem ;
  }
  for (Index_t i = 1; i < threads; ++i) {
    if(dthydro_per_thread[i] < dthydro_per_thread[0]) {
      dthydro_per_thread[0] = dthydro_per_thread[i];
      hydro_elem_per_thread[0] = hydro_elem_per_thread[i];
    }
  }
  if (hydro_elem_per_thread[0] != -1) {
    dthydro = dthydro_per_thread[0] ;
  }
  return ;
}
```

C++ with OpenMP

C++ Standard Parallelism

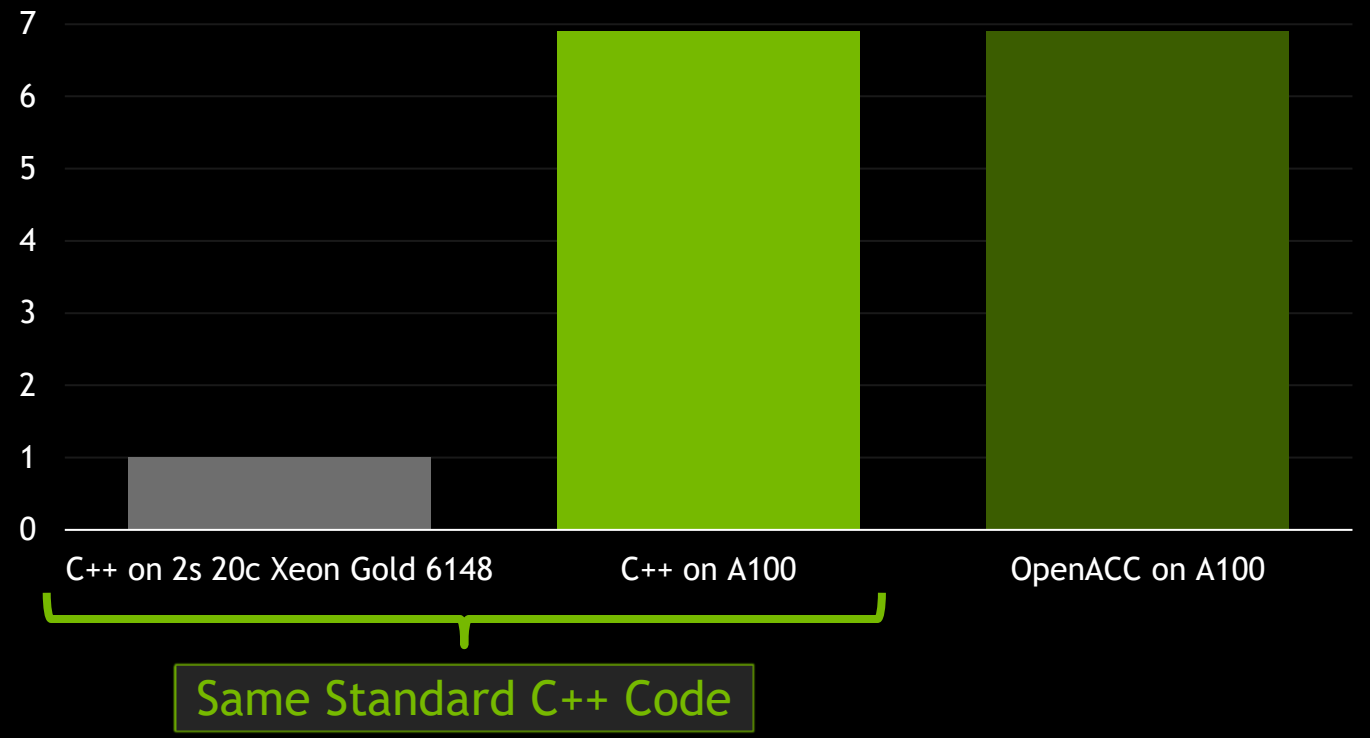
- Composable, compact and elegant.
- Easy to read and maintain.
- ISO Standard.
- Portable – NVC++, GCC, ICPC, MSVC, …

```cpp
static inline void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                                               Index_t *regElemlist,
                                               Real_t dvovmax,
                                               Real_t &dthydro)
{
  dthydro = std::transform_reduce(
    std::execution::par, counting_iterator(0), counting_iterator(length),
    dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
    [=, &domain](Index_t i)
  {
      Index_t indx = regElemlist[i];
      if (domain.vdov(indx) == Real_t(0.0)) {
        return std::numeric_limits<Real_t>::max();
      } else {
        return dvovmax / (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
      }
  });
}
```

Standard C++

LULESH Performance

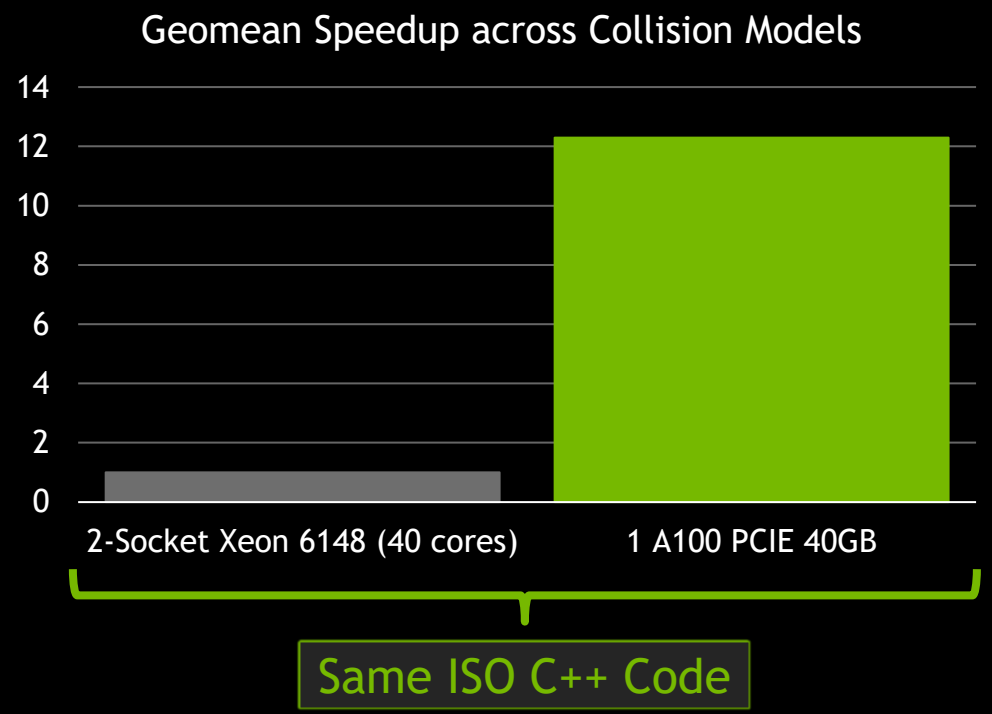
Speedup – Higher is Better



NVIDIA.

STLBM

Parallel Lattice Boltzmann with C++ Parallel Algorithms



- Framework for parallel lattice-Boltzmann simulations on multiple platforms, including many-core CPUs and GPUs.
- Implemented with Standard C++ Parallel Algorithms to achieve parallel efficiency.
- No language extensions, external libraries, vendor-specific code annotations, or pre-compilation steps.

*"We have with delight discovered the NVIDIA "stdpar" implementation of Standard C++ Parallel Algorithms. … We believe that the result produces state-of-the-art performance, is highly didactical, and introduces **a paradigm shift in cross-platform CPU/GPU programming** in the community."*

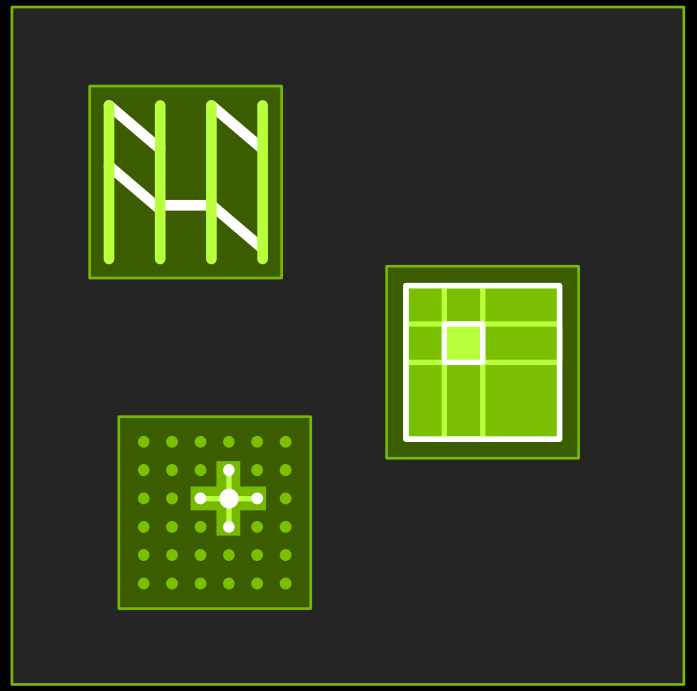
— Professor Jonas Latt, University of Geneva

https://gitlab.com/unigehpfs/stlbm

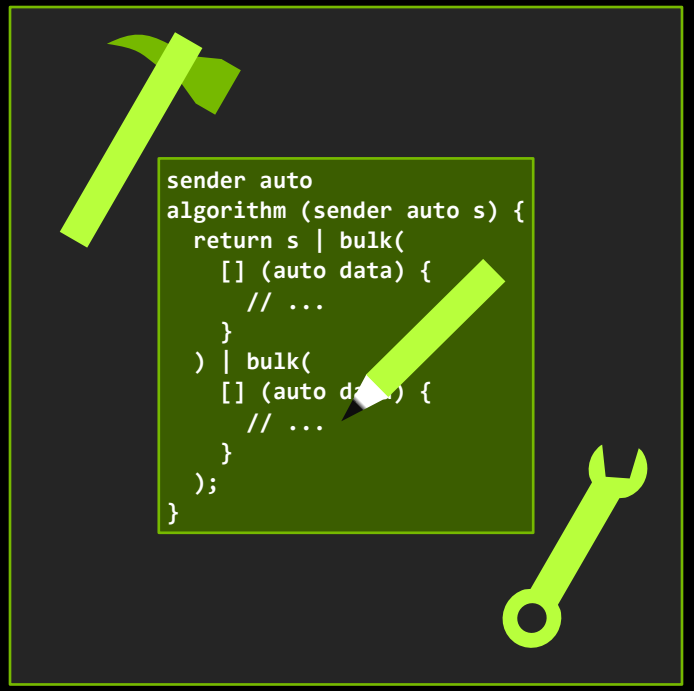
NVIDIA.

Standard C++ Parallelism

Common Algorithm that Dispatch to Vendor-Optimized Parallel Libraries

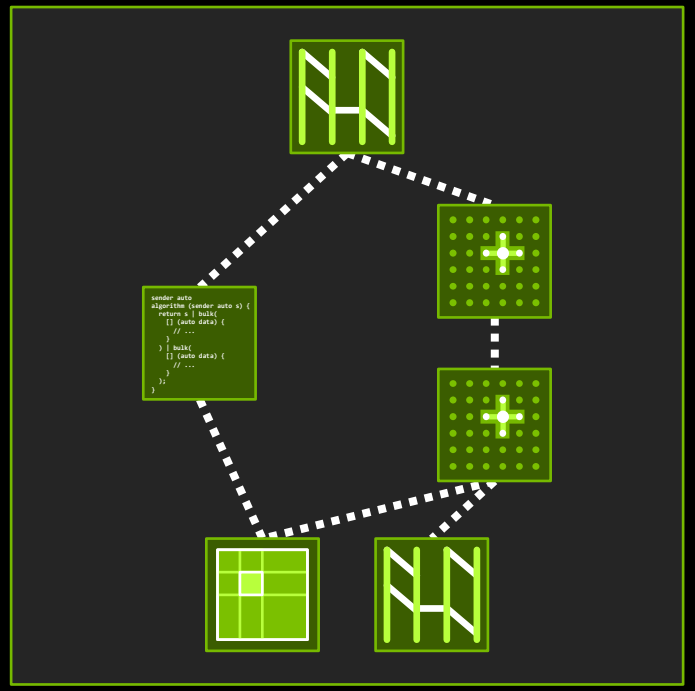


Tools to Write Your Own Parallel Algorithms that Run Anywhere



```
sender auto
algorithm (sender auto s) {
  return s | bulk(
    [] (auto data) {
      // ...
    }
  ) | bulk(
    [] (auto data) {
      // ...
    }
  );
}
```

Mechanisms for Composing Parallel Invocations into Task Graphs



Standard C++ Parallelism

C++17

Parallel Algorithms

- In NVC++
- Parallel and vector concurrency

Forward Progress Guarantees

- Extend the C++ execution model for accelerators

Memory Model Clarifications

- Extend the C++ memory model for accelerators

C++20

Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators
- In libcu++:
  - `std::atomic<T>`
  - `std::barrier`
  - `std::counting_semaphore`
  - `std::atomic<T>::wait/notify_*`
  - `std::atomic_ref<T>`

C++23 and Beyond

Executors

- Simplify launching and managing parallel work across CPUs and accelerators

`std::mdspan/mdarray`

- HPC-oriented multi-dimensional array abstractions.

Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries

Extended Floating Point Types

- First-class support for formats new and old: `std::float16_t/float64_t`

Standard C++ Executors

```
std::vector<double> data = /* ... */;

auto s = std::just_on(cuda::scheduler, data)
       | std::bulk(N,
           [](auto idx, auto& v)
           { if (idx > 0) v[idx] += v[idx - 1]; })
       | std::bulk(N,
           [](auto idx, auto& v)
           { 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 C++ Executors

operator| composes parallel work together.
f | g | h == h(g(f))

```
data = /* ... */;

auto s = std::just_on(cuda::scheduler, data)
       | std::bulk(N,
           [](auto idx, auto& v)
           { if (idx > 0) v[idx] += v[idx - 1]; })
       | std::bulk(N,
           [](auto idx, auto& v)
           { 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 C++ Executors

`operator|` composes parallel work together.
`f | g | h == h(g(f))`

Schedulers abstract execution resources, allowing us to write generic parallel algorithms.

```
data = /
auto s = std::just_on(cuda::scheduler, data)
       | std::bulk(N,
           [](auto idx, auto& v)
           { if (idx > 0) v[idx] += v[idx - 1]; })
       | std::bulk(N,
           [](auto idx, auto& v)
           { 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 C++ Executors

operator| composes parallel work together.
`f | g | h == h(g(f))`

Schedulers abstract execution resources, allowing us to write generic parallel algorithms.

```
data = /
auto s = std::just_on(cuda::scheduler, data)
       | std::bulk(N,
           [](auto idx, auto& v)
           { if (idx > 0) v[idx] +=
       | std::bulk(N,
           [](auto idx, auto& v)
           { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

lazy and

to author C++ that em.

Pipable sender algorithms	
`sender just(T t);`	Pass the value t to the next work item.
`sender on(sender last, scheduler s);`	Switch to scheduler s for the next work item.
`sender then(sender last, invocable f);`	Call f with the value sent by last.
`sender bulk(sender last, shape n, invocable f);`	Spawn n tasks that call f.

Standard C++ Executors

`operator|` composes parallel work together.
`f | g | h == h(g(f))`

Schedulers abstract execution resources, allowing us to write generic parallel algorithms.

```
data = /
auto s = std::just_on(cuda::scheduler, data)
       | std::bulk(N,
           [](auto idx, auto& v)
           { if (idx > 0) v[idx] +=
       | std::bulk(N,
           [](auto idx, auto& v)
           { v[idx] *= v[idx]; });

std::sync_wait(s);
```

Executors are a portable abstraction for composing and

lazy and

to author C++ that em.

Pipable sender algorithms	
`sender just(T t);`	Pass the value t to the next work item.
`sender on(sender last, scheduler s);`	Switch to scheduler s for the next work item.
`sender then(sender last, invocable f);`	Call f with the value sent by last.
`sender bulk(sender last, shape n, invocable f);`	Spawn n tasks that call f.

Standard C++ Linear Algebra

```
std::mdspan A(/* ... */, num_rows, num_cols);
std::mdspan x(/* ... */, num_cols);
std::mdspan y(/* ... */, num_rows);

// 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);
```

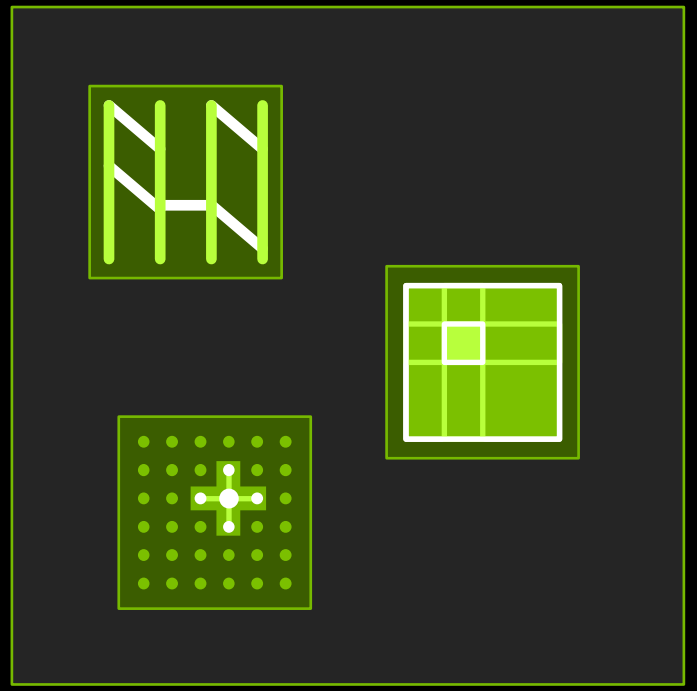
A modern Standard C++ abstraction based on the BLAS.

It provides a portable interface to both sequential and parallel linear algebra libraries on whatever platform you are on.

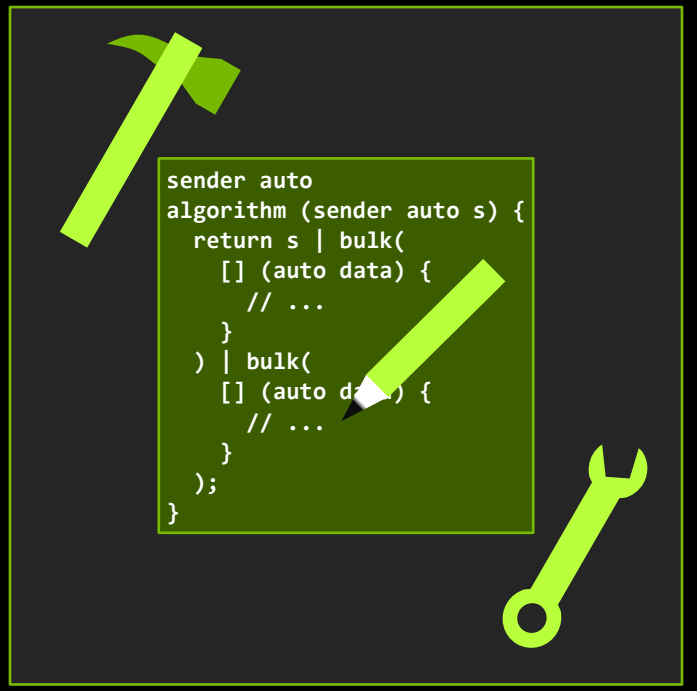
`std::mdspan`: non-owning multi-dimensional span type.

Standard C++ Parallelism

Common Algorithm that Dispatch to Vendor-Optimized Parallel Libraries

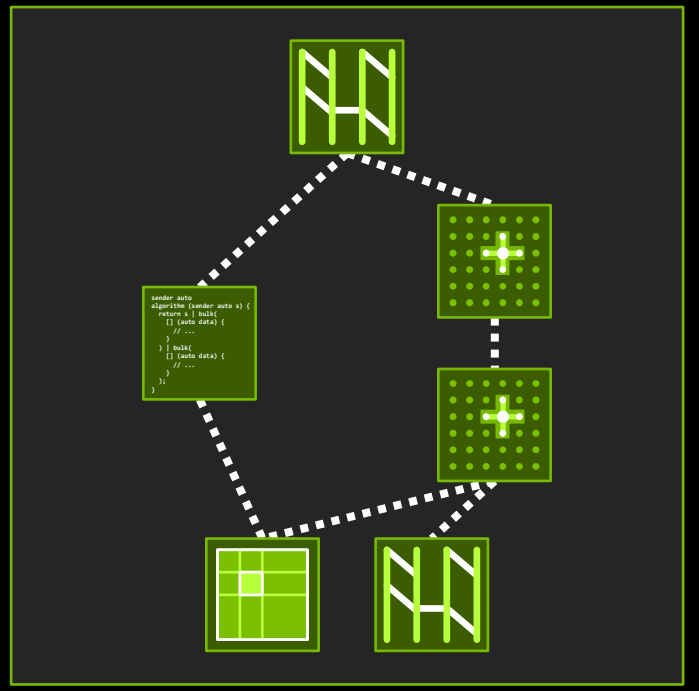


Tools to Write Your Own Parallel Algorithms that Run Anywhere

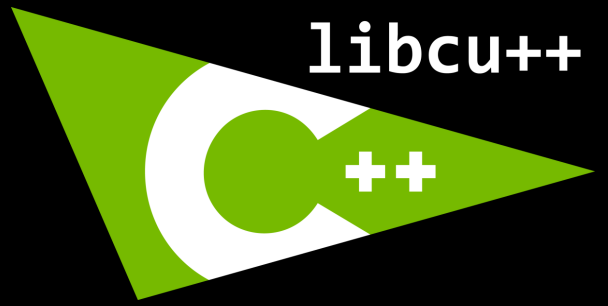


```
sender auto
algorithm (sender auto s) {
  return s | bulk(
    [] (auto data) {
      // ...
    }
  ) | bulk(
    [] (auto data) {
      // ...
    }
  );
}
```

Mechanisms for Composing Parallel Invocations into Task Graphs

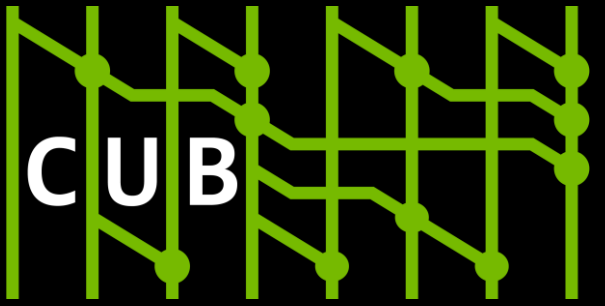


C++ Core Compute Libraries

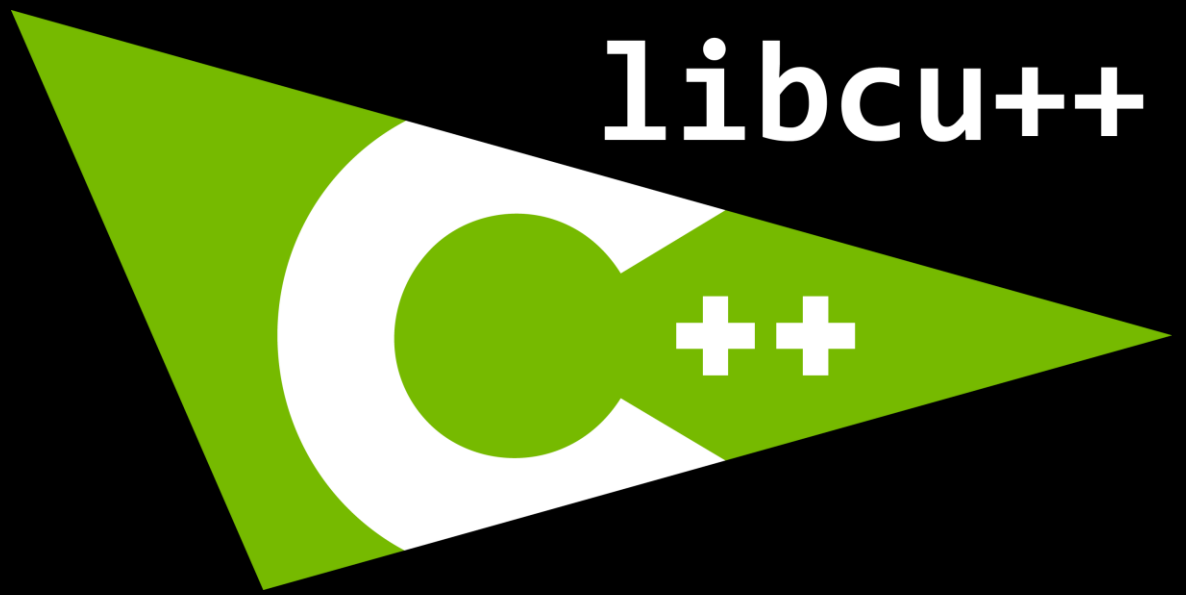


The Standard Library for your entire system
https://github.com/NVIDIA/libcudacxx

The C++ parallel algorithms library
https://github.com/NVIDIA/thrust



Cooperative primitives for CUDA C++
https://github.com/NVIDIA/cub



The opt-in, heterogeneous, incremental CUDA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

Opt-in

Does not interfere with or replace your host standard library.

```
// ISO C++, __host__ only.
#include <atomic>
std::atomic<int> x;

// CUDA C++, __host__ __device__.
// Strictly conforming to the ISO C++.
#include <cuda/std/atomic>
cuda::std::atomic<int> x;

// CUDA C++, __host__ __device__.
// Conforming extensions to ISO C++.
#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

Heterogeneous

Copyable/Movable objects can migrate between host & device.

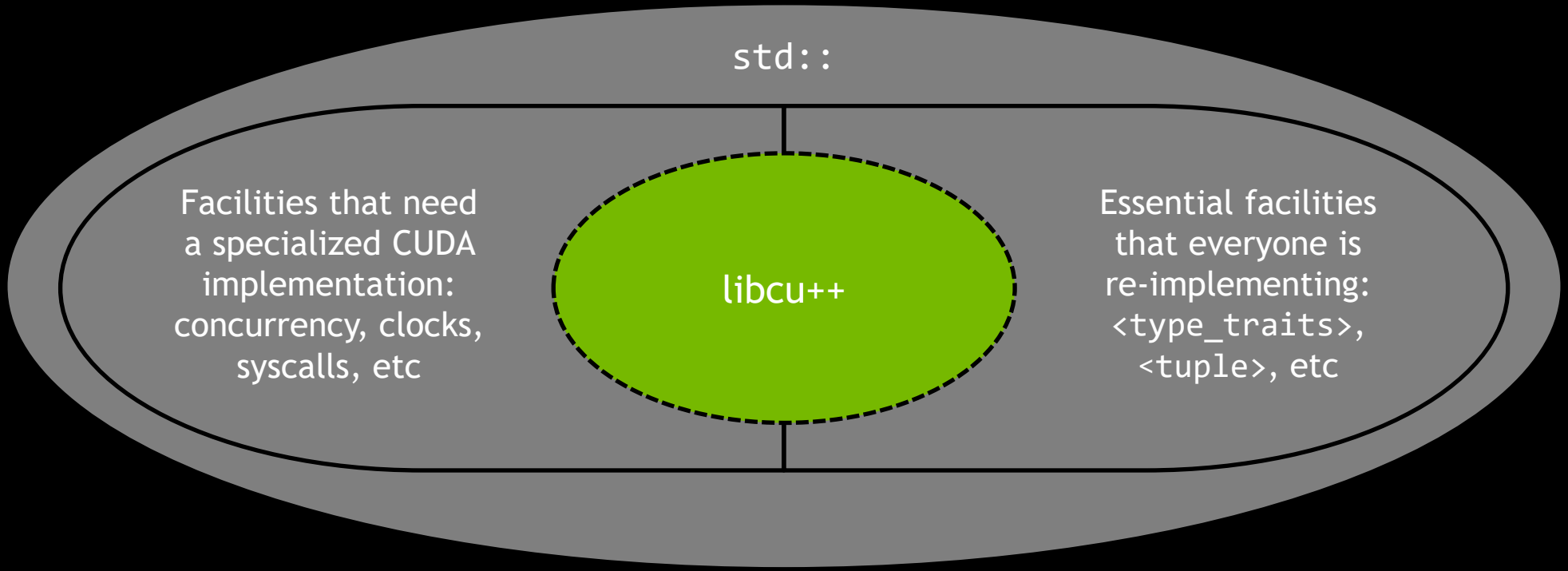
Host & device can call all (member) functions.

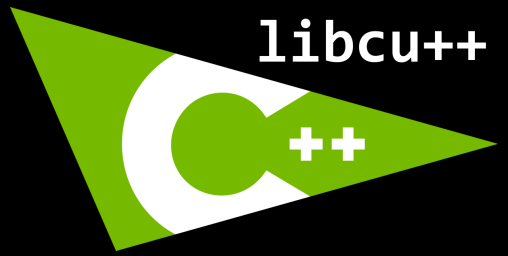
Host & device can concurrently use synchronization primitives*.

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

Incremental

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





CUDA 10.2	CUDA 11.x	Future
<cuda/[std/]atomic> (SM60+) <cuda/std/type_traits>	atomic::wait/notify (SM70+) <cuda/[std/]barrier> (SM70+) memcpy_async (SM70+) <cuda/std/latch> (SM70+) <cuda/std/semaphore> (SM70+) <cuda/std/chrono> <cuda/std/ratio> <cuda/std/tuple> (11.2) <cuda/std/complex> (11.3)	CUDA Memory Resources Async Allocation, Pools CUDA Stream Abstractions atomic_ref <cuda/std/mutex> <cuda/std/array> <cuda/std/utility> <cuda/std/filesystem> String Processing Iostreams

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

} // namespace cuda
```

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

  return data;
}
```

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

  return data;
}
```

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

```
__host__ __device__
int poll_flag_then_read(int& flag, int& data) {


  while (1 != atomicAdd(flag, 0)) // <- Should be atomic load.
    ; // <- Spinloop without backoff is bad under contention.
  __threadfence_system(); // <- 9 out of 10 of you forget this one!
  return data;
}
```

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

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

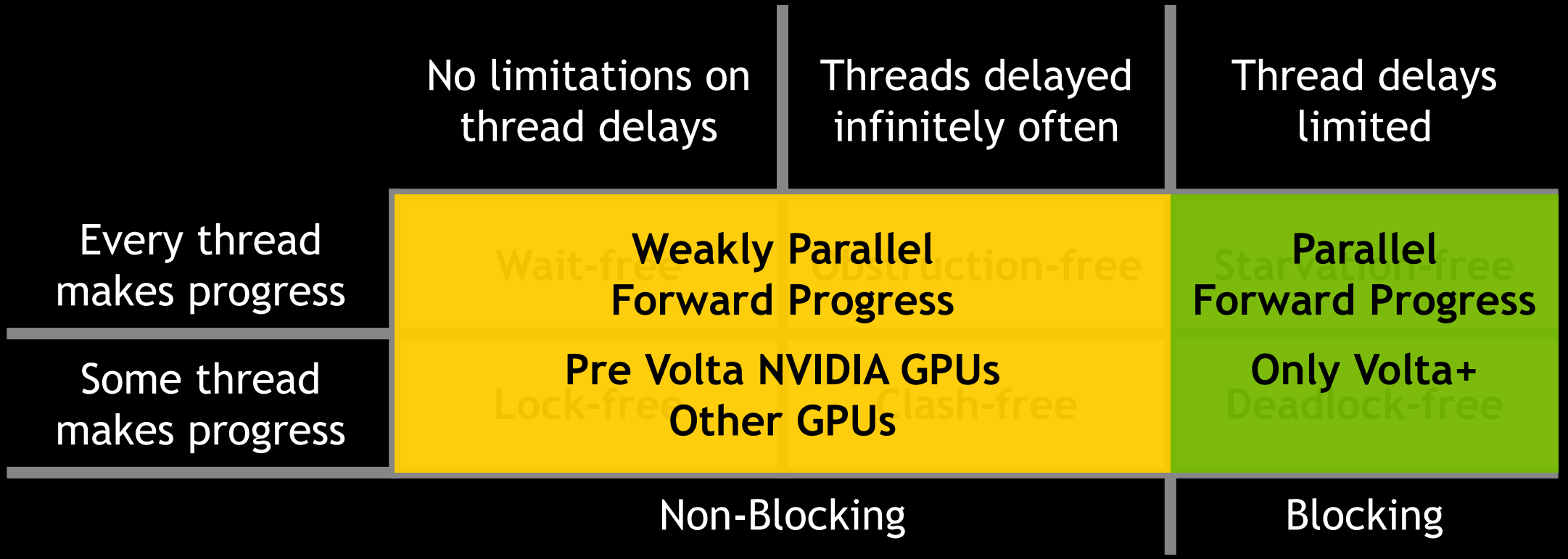
```
__host__ __device__
int poll_flag_then_read_excellent(atomic<bool>& flag, int& data) {
  flag.wait(false, memory_order_acquire);
  // ^^^ Backoff to mitigate heavy contention.
  return data;
}
```

Volta+ NVIDIA GPUs deliver and libcu++ exposes:

C++ Parallel Forward Progress Guarantees.

The C++ Memory Model.

Why does this matter?

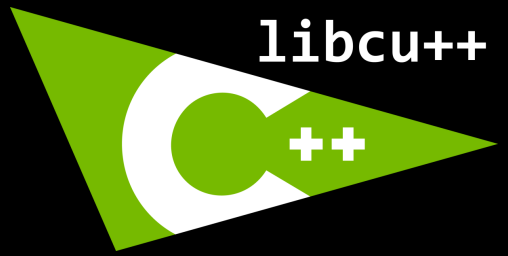


Source: http://www.cs.tau.ac.il/~shanir/progress.pdf

Why does this matter?

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

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



The opt-in, heterogeneous, incremental CUDA C++ standard library.

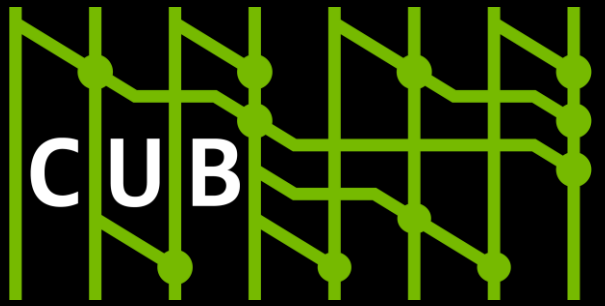
https://github.com/NVIDIA/libcudacxx

CUDA 10.2	CUDA 11.x	Future
<cuda/[std/]atomic> (SM60+) <cuda/std/type_traits>	atomic::wait/notify (SM70+) <cuda/[std/]barrier> (SM70+) memcpy_async (SM70+) <cuda/std/latch> (SM70+) <cuda/std/semaphore> (SM70+) <cuda/std/chrono> <cuda/std/ratio> <cuda/std/tuple> (11.2) <cuda/std/complex> (11.3)	CUDA Memory Resources Async Allocation, Pools CUDA Stream Abstractions atomic_ref <cuda/std/mutex> <cuda/std/array> <cuda/std/utility> <cuda/std/filesystem> String Processing Iostreams

Latest Work:

- Asynchronous Interfaces.
- Unified Memory Abstractions.
- Memory Resources and Pools.
- Modernization: C++11 only.

```
std::vector<float> h;
thrust::vector<float> d;

thrust::event e = thrust::async::copy(par, h.begin(), h.end(), d.begin());
thrust::future<float> f = thrust::async::reduce(par.after(e), y.begin(), y.end());
```



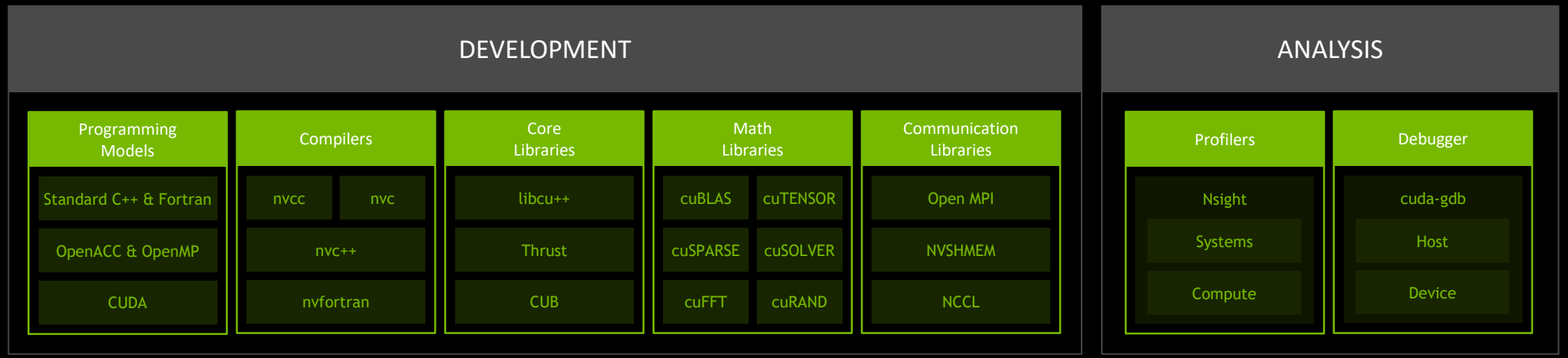
Latest Work:

- Performance Improvements.
- New Algorithms.
- Productization (CUDA 11.0).
- Modernization: C++11 only.

```
template <typename R>
__global__ void reduce(R, …) {
  __shared__ typename R::TempStorage tmp;
  int local_s = …;
  int s = R(tmp).Sum(local_s);
}

reduce<<<32, 1>>>(cub::WarpReduce<int>{}, …);
reduce<<<128, 1>>>(cub::BlockReduce<int, 128>{}, …);
```

The NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, and in the Cloud



Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect

HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA

7-8 Releases Per Year | Freely Available