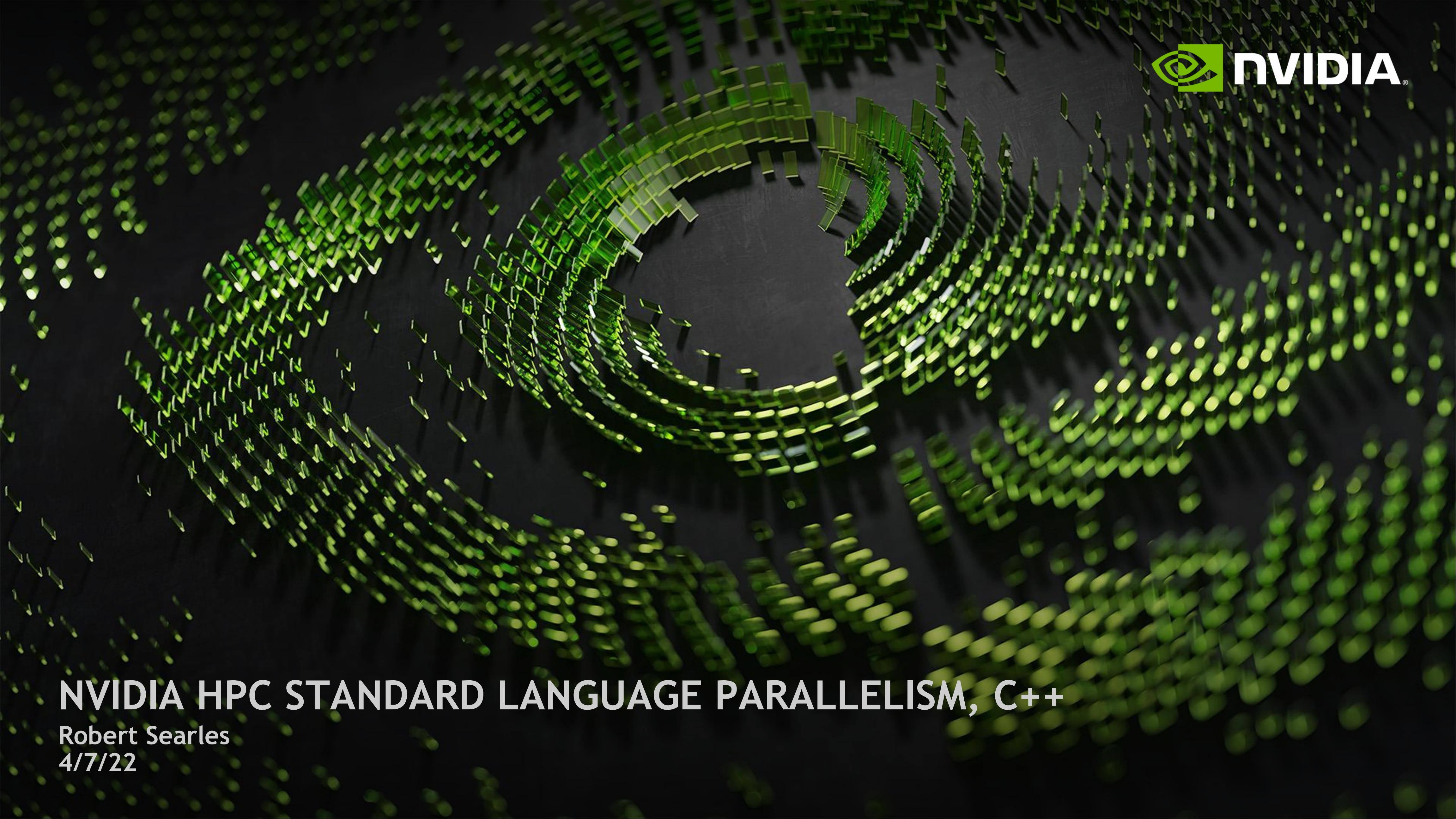


NVIDIA HPC STANDARD LANGUAGE PARALLELISM, C++

Robert Searles
4/7/22

PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
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
```

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}

#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__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, ...);
```

ACCELERATION LIBRARIES

Core | Math | Communication | Data Analytics | AI | Quantum

ACCELERATED STANDARD LANGUAGES

Parallel performance for wherever your code runs

ISO C++

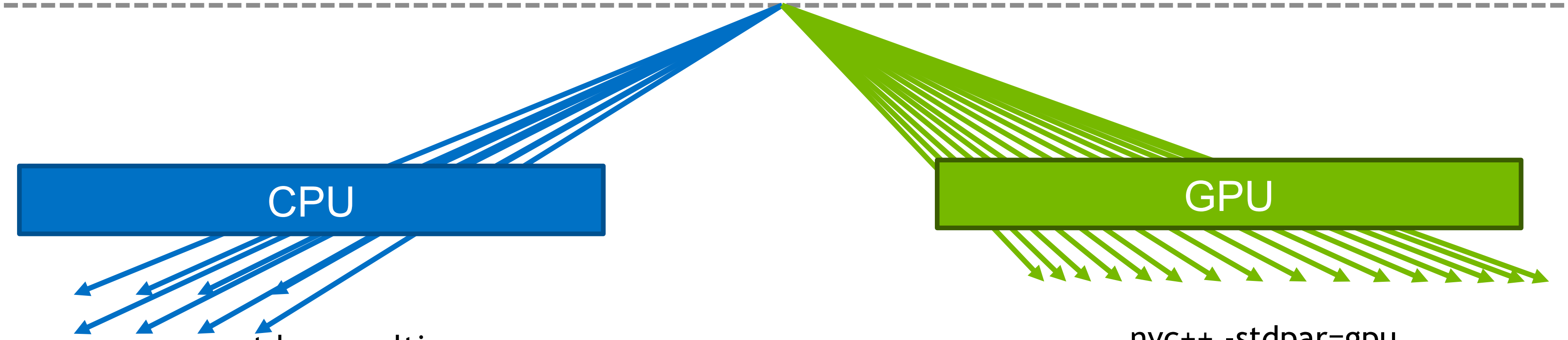
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

ISO Fortran

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

Python

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```



nvc++ -stdpar=multicore
nvfortran -stdpar=multicore
legate -cpus 16 saxpy.py

nvc++ -stdpar=gpu
nvfortran -stdpar=gpu
legate -gpus 1 saxpy.py

FUTURE OF CONCURRENCY AND PARALLELISM IN HPC: STANDARD LANGUAGES

How did we get here?

ON-GOING LONG-TERM INVESTMENT

ISO committee participation from industry, academia and government labs.

Fruit born in 2020 was planted over the previous decade.

Focus on enhancing concurrency and parallelism for all.

Open collaboration between partners and competitors.

Past investments in directives enabled rapid progress.

MAJOR FEATURES

Memory Model Enhancements

C++14 Atomics Extensions

C++17 Parallel Algorithms

C++20 Concurrency Library

C++23 Multi-Dim. Array Abstractions

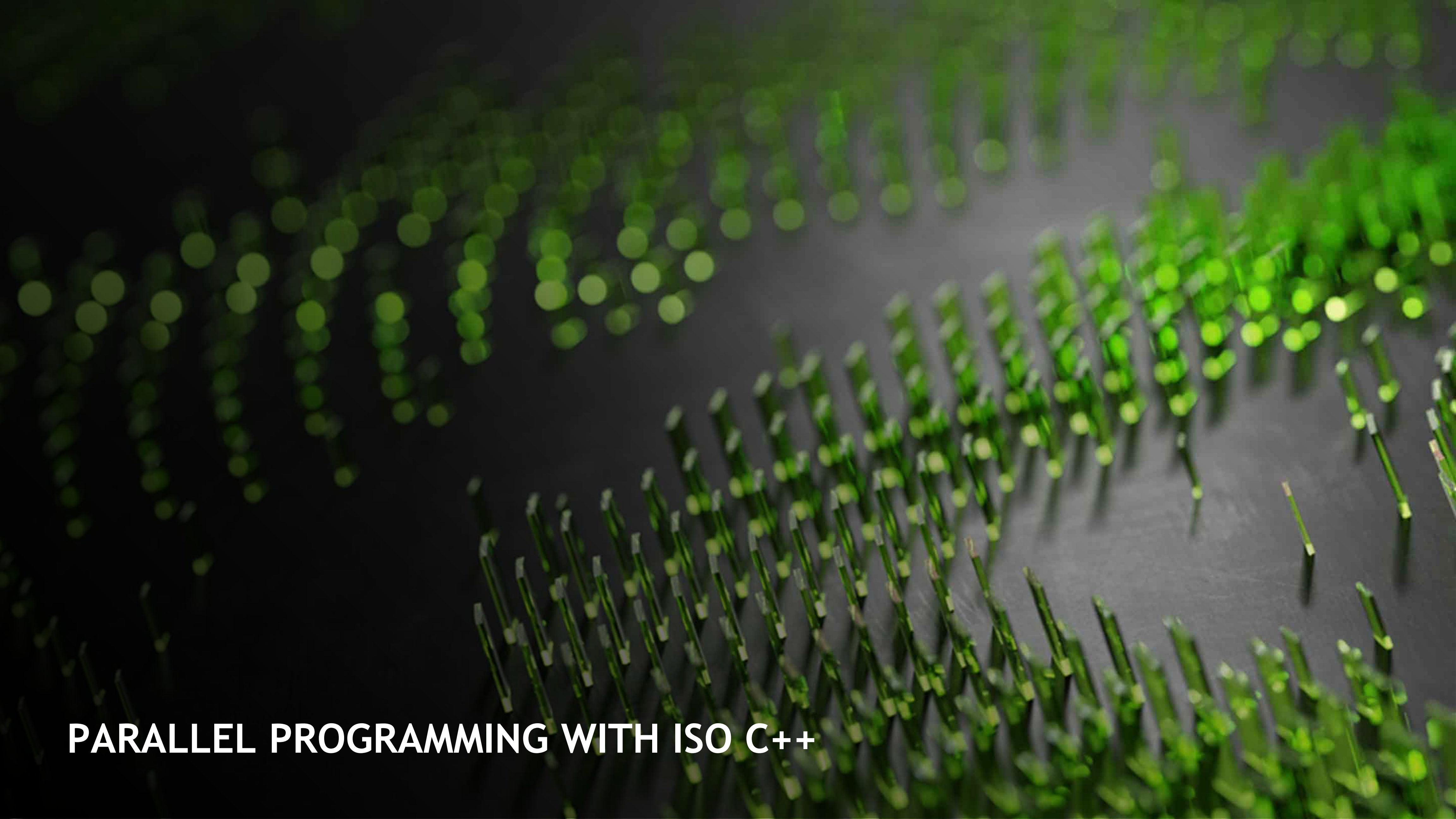
C++23 Extended Floating Point Types

C++23 Range Based Parallel Algorithms

C++2X Executors

C++2X Linear Algebra

Fortran 202X DO CONCURRENT Reduction



PARALLEL PROGRAMMING WITH ISO C++

HPC PROGRAMMING IN ISO C++

C++ Parallel Algorithms

```
std::sort(std::execution::par, c.begin(), c.end());

std::unique(std::execution::par, c.begin(), c.end());
```

- Introduced in C++17
- Parallel and vector concurrency via execution policies

  `std::execution::par, std::execution::par_unseq, std::execution::seq`
- Several new algorithms in C++17 including

  `std::for_each_n(POLICY, first, size, func)`
- Insert std::execution::par as first parameter when calling algorithms
- NVC++ (since 20.7): automatic CPU or GPU acceleration of C++17 parallel algorithms
  - Leverages CUDA Unified Memory

HPC PROGRAMMING IN ISO C++

ISO is the place for portable concurrency and parallelism

C++17 & C++20

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

Ranges

- Simplifies iterating over a range of values

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>`

Preview support coming to NVC++

C++23

`std::mdspan/mdarray`

- HPC-oriented multi-dimensional array abstractions.

Range-Based Parallel Algorithms

- Improved multi-dimensional loops

Extended Floating Point Types

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

And Beyond

Executors / Senders-Recievers

- Simplify launching and managing parallel work across CPUs and accelerators

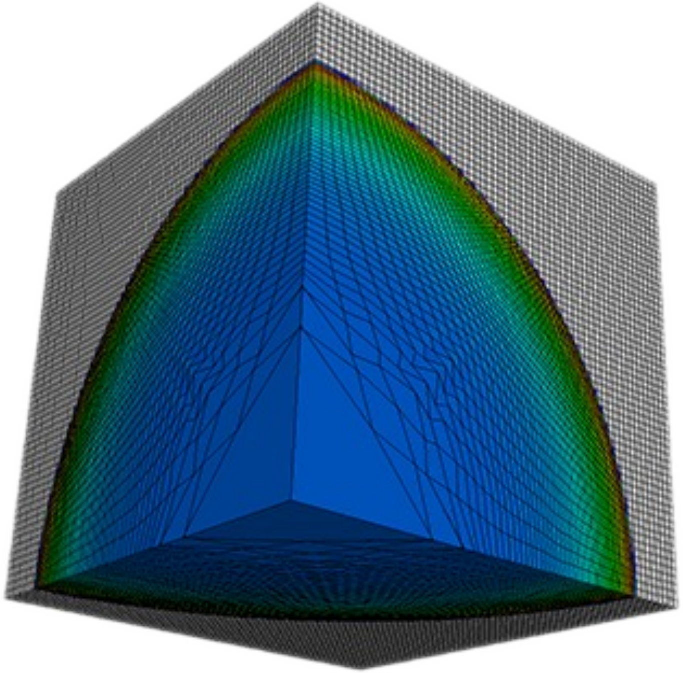
Linear Algebra

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

C++17 PARALLEL ALGORITHMS

Lulesh Hydrodynamics Mini-app

- ~9000 lines of C++
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, ISO C++…
- Designed to stress compiler vectorization, parallel overheads, on-node parallelism



codesign.llnl.gov/lulesh

STANDARD C++

- Composable, compact and elegant
- Easy to read and maintain
- ISO Standard
- Portable – nvc++, g++, icpc, MSVC, …

```
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 ;
}
```

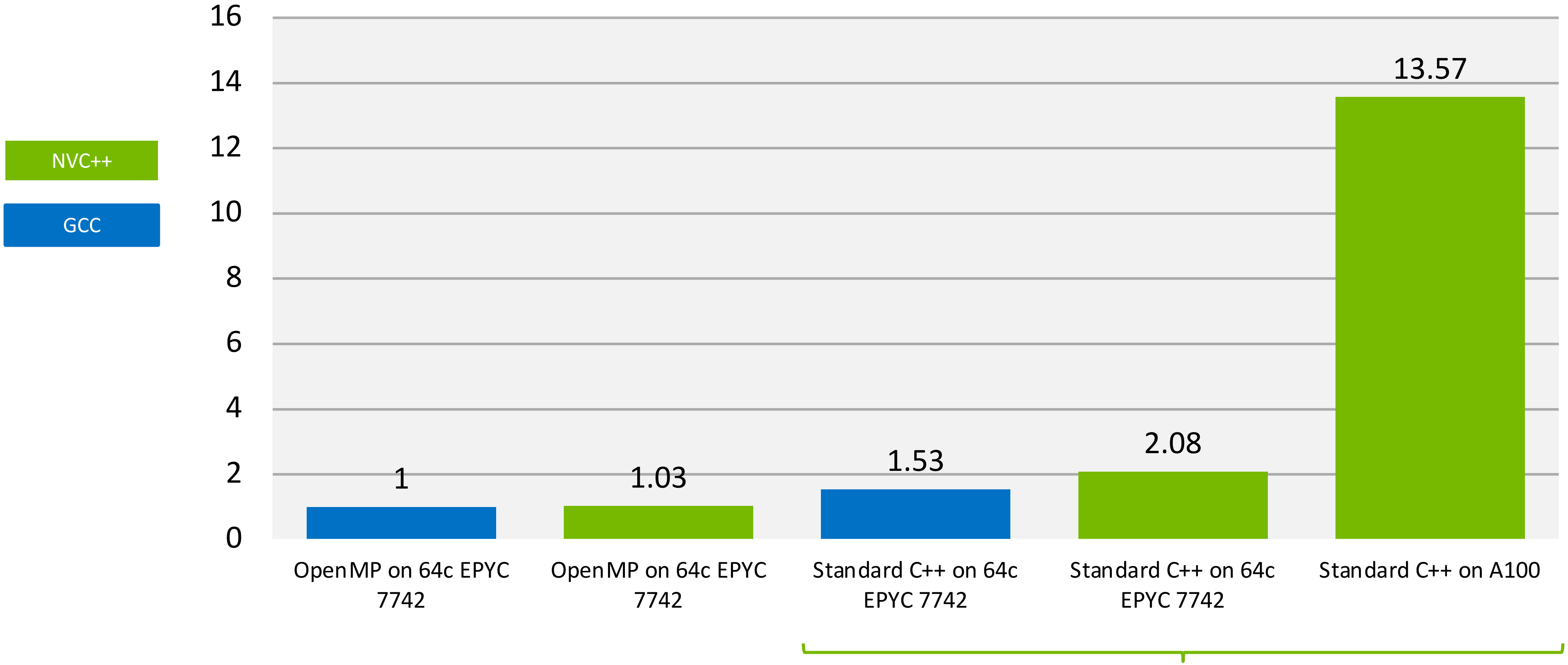
C++ with OpenMP

```
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++

C++ STANDARD PARALLELISM

Lulesh Performance

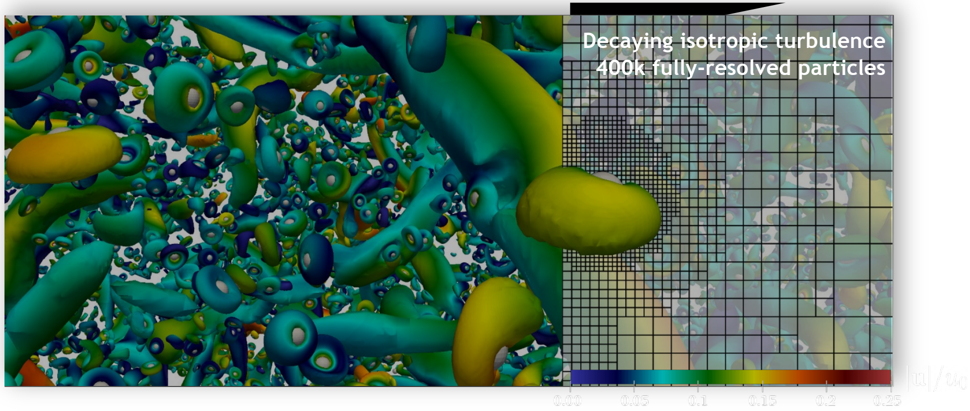


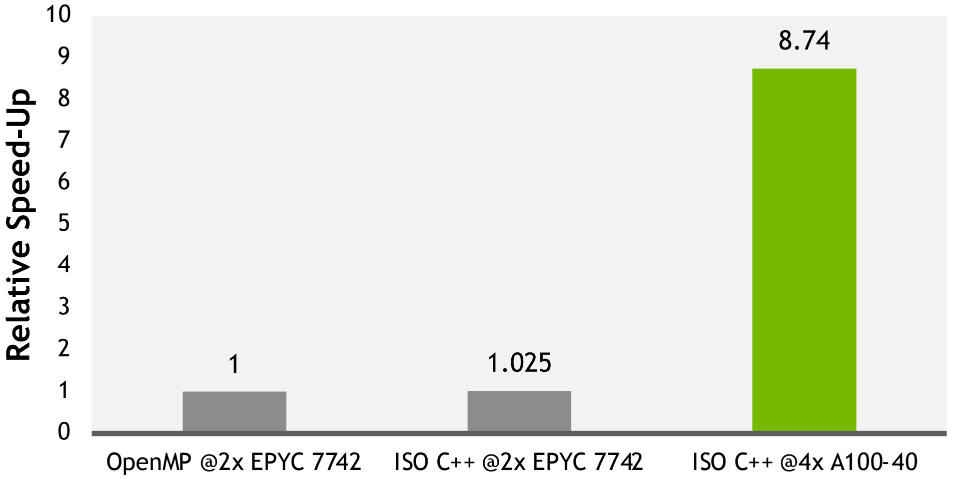
Same ISO C++ Code

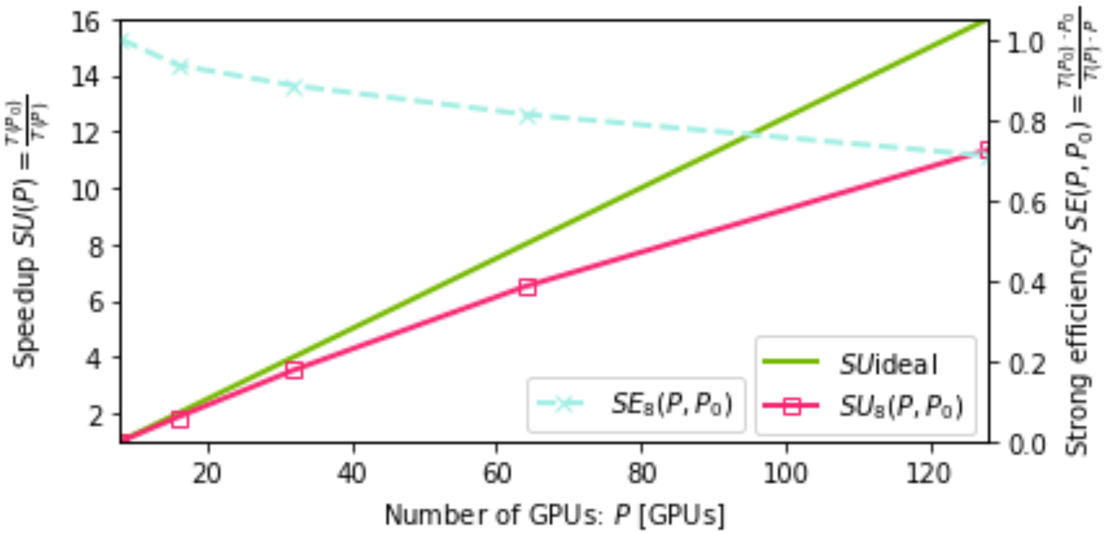
M-AIA

Multi-physics simulation framework developed at the Institute of Aerodynamics, RWTH Aachen University

- Hierarchical grids, complex moving geometries
- Adaptive meshing, load balancing
- Numerical methods: FV, DG, LBM, FEM, Level-Set, ...
- Physics: aeroacoustics, combustion, biomedical, ...
- Developed by ~20 PhDs (Mech. Eng.), ~500k LOC++
- **Programming model:** MPI + ISO C++ parallelism







M-AIA REFACTORING

```
#pragma omp parallel // OpenMP parallel region
{
  #pragma omp for // OpenMP for loop
  for (MInt i = 0; i < noCells; i++) { // Loop over all cells
    if (timeStep % ipow2[maxLevel_ - clevel[i * distLevel]] == 0) { // Multi-grid loop
      const MInt distStartId = i * nDist; // More offsets for 1D accesses // Local offsets
      const MInt distNeighStartId = i * distNeighbors;
      const MFloat* const distributionsStart = &[distributions[distStartId];
      for (MInt j = 0; j < nDist - 1; j += 2) { // Unrolled loop distributions (factor 2)
        if (neighborId[I * distNeighbors + j] > -1) { // First unrolled iteration
          const MInt n1StartId = neighborId[distNeighStartId + j] * nDist;
          oldDistributions[n1StartId + j] = distributionsStart[j]; // 1D access AoS format
        }
        if (neighborId[I * distNeighbors + j + 1] > -1) { // Second unrolled iteration
          const MInt n2StartId = neighborId[distNeighStartId + j + 1] * nDist;
          oldDistributions[n2StartId + j + 1] = distributionsStart[j + 1];
        }
      }
      oldDistributions[distStartId + lastId] = distributionsStart[lastId]; // Zero-th
distribution
    }
  }
}
```

Single ISO C++ code that runs in parallel on CPU and GPU

```
std::for_each_n(par_unseq, start, noCells, [=](auto i) { // Parallel for
  if (timeStep % IPOW2[maxLevel_ - a_level(i)] != 0) // Multi-level loop
    return;
  for (MInt j = 0; j < nDist; ++j) {
    if (auto n = c_neighborId(i, j); n == -1) continue;
    a_oldDistribution(n, j) = a_distribution(i, j); // SoA or AoS mem_fn
  }
});
```

STLBM

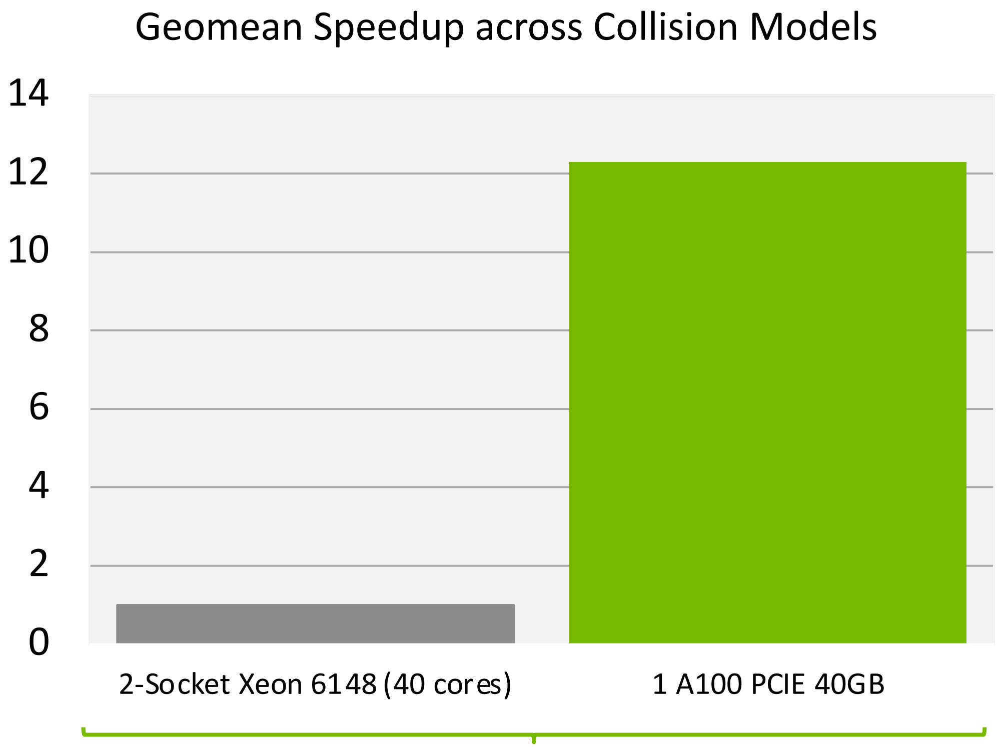
Many-core Lattice Boltzmann with C++ Parallel Algorithms

- Framework for parallel lattice-Boltzmann simulations on multiple platforms, including many-core CPUs and GPUs
- Implemented with C++17 standard (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 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
https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32076/
https://www.nvidia.com/en-us/on-demand/session/gtcfall21-a31329



Same ISO C++ Code

C++ PARALLEL ALGORITHM REFACTORING

Simple loops may be replaced with a for_each_n or transform algorithm.

```
#pragma omp parallel for firstprivate(qlc_monoq, \
                qqc_monoq, monoq_limiter_mult, \
                monoq_max_slope, ptiny)
for ( Index_t i = 0 ; i < domain.regElemSize(r); ++i ) {
 …
}
```

```
std::for_each_n(
    std::execution::par, counting_iterator(0), \
    domain.regElemSize(r),
    [=, &domain](Index_t i) {
    …
}
```

```
std::transform(
    std::execution::par, counting_iterator(0), \
    domain.regElemSize(r),
    [=, &domain](Index_t i) {
    …
}
```

C++ PARALLEL ALGORITHM REFACTORING

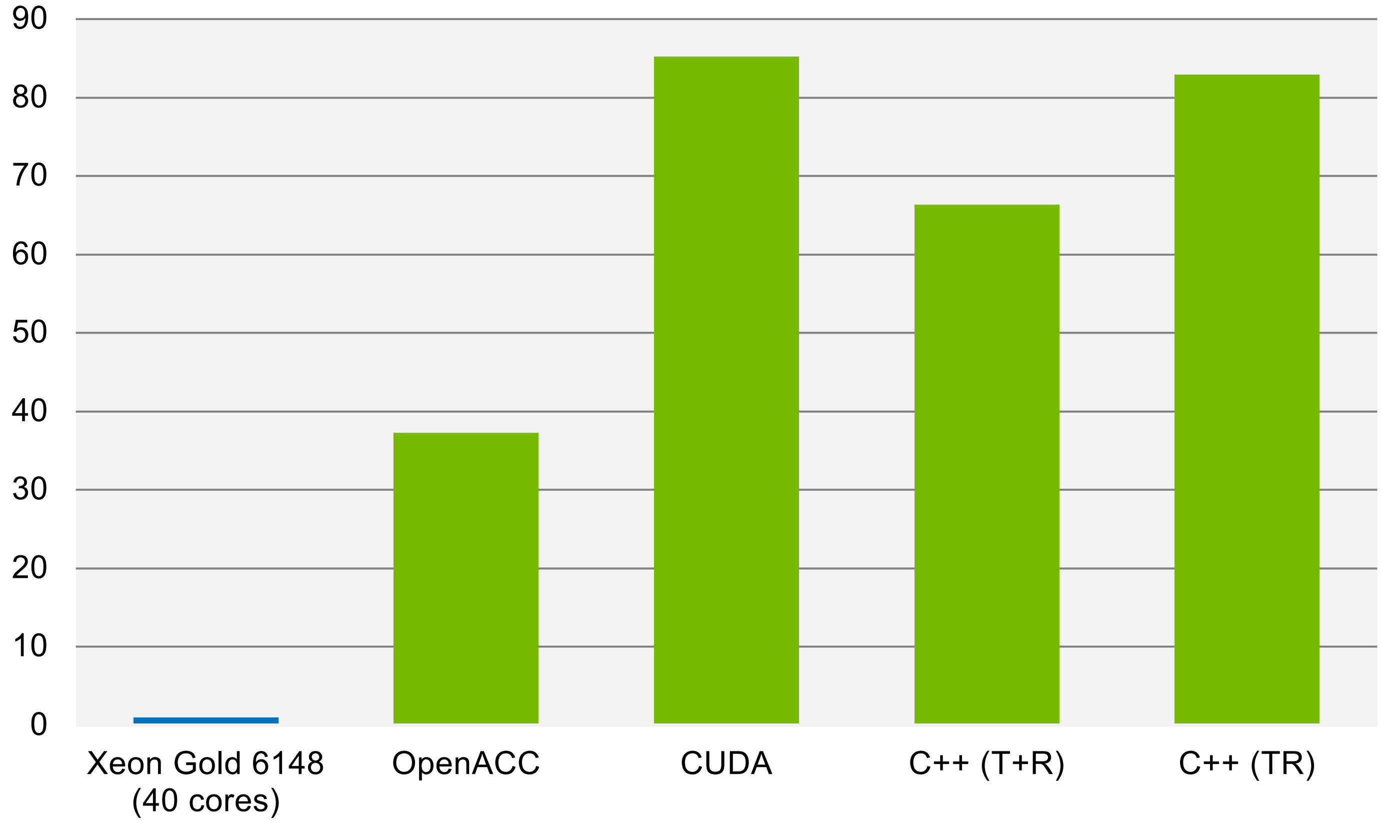
When possible, use the algorithm that fits your use case. E.g. a transform_reduce enables parallel execution with a reduction.

```
#pragma acc parallel loop default(present) deviceptr(d_normals)
  for (unsigned i = 0; i < N_PATHS; i++) {
    float s_curr = S0;
    for (unsigned n = 0; n < N_STEPS; n++) {
      s_curr +=
          tmp1 * s_curr + sigma * s_curr * tmp3 *
          d_normals[i + n * N_PATHS];
      running_average += (s_curr - running_average) /
                         (n + 1.0);
      if (running_average <= B) {
        break;
      }
    }

    float payoff = (running_average > K ?
                    running_average - K : 0.f);
    d_s[i] = tmp2 * payoff;
  }

#pragma acc parallel loop default(present) reduction(+ : sum)
  for (unsigned i = 0; i < N_PATHS; ++i) {
    sum += (double)d_s[i];
  }
```

```
sum = std::transform_reduce(
      std::execution::par, counting_iterator(0),
                           counting_iterator(N_PATHS),
      0.0, std::plus<>(), [=](int i) {
        double running_average = 0.0;
        float s_curr = S0;
        for (unsigned n = 0; n < N_STEPS; n++) {
          s_curr += tmp1 * s_curr +
                    sigma * s_curr * tmp3 * d_normals[i + n*N_PATHS];
          running_average += (s_curr - running_average) / (n + 1.0);
          if (running_average <= B) {
            break;
          }
        }

        double payoff = (running_average > K ?
                         running_average - K : 0.f);
        return (tmp2 * payoff);
      });
```

USING THE RIGHT ALGORITHM

Speedup – Higher is Better



USING C++ RANGES

Simplify iterating over a range of values

- Ranges provides a standardized way to specify the range of values over which to iterate.
- The iota view is a simple way to declare the iteration space of the loop you have converted into a parallel algorithm.
- To use iota you must have GCC10 or newer installed and use specify –std=c++20 in your nvc++ options

```
auto v = std::ranges::views::iota(0, N*M);

std::for_each(

  std::execution::par_unseq,

  begin(v), end(v),

  […] (auto idx) { … });
```

*Prior to nvc++ 22.2 or if GCC10 is not available, the Thrust counting_iterator should be used in place of iota.

USING C++ RANGES

Simplify iterating over a range of values

- Ranges provides a standardized way to specify the range of values over which to iterate.
- The iota view is a simple way to declare the iteration space of the loop you have converted into a parallel algorithm.
- To use iota you must have GCC10 or newer installed and use specify –std=c++20 in your nvc++ options

```
auto v = std::ranges::views::iota(0, N*M);
std::for_each(
  std::execution::par_unseq,
  begin(v), end(v),
  […] (auto idx) { … });
```

- C++23 is expected to add the cartesian_product view, which expands ranges to support multi-dimension ranges.

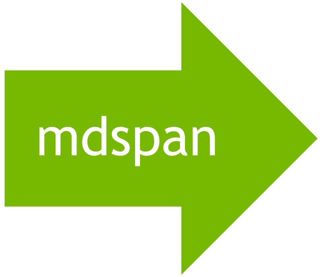
```
auto v =
std::ranges::views::cartesian_product(
  stdv::iota(0, N),
  stdv::iota(0, M));

std::for_each(std::execution::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i,j] = idx;
    ...
  });
```

*Prior to nvc++ 22.2 or if GCC10 is not available, the Thrust counting_iterator should be used in place of iota.

WHAT'S COMING IN C++: MDSPAN

C++23 brings mdspan, a standardized way to access multi-dimensional data, which composes well with ranges.

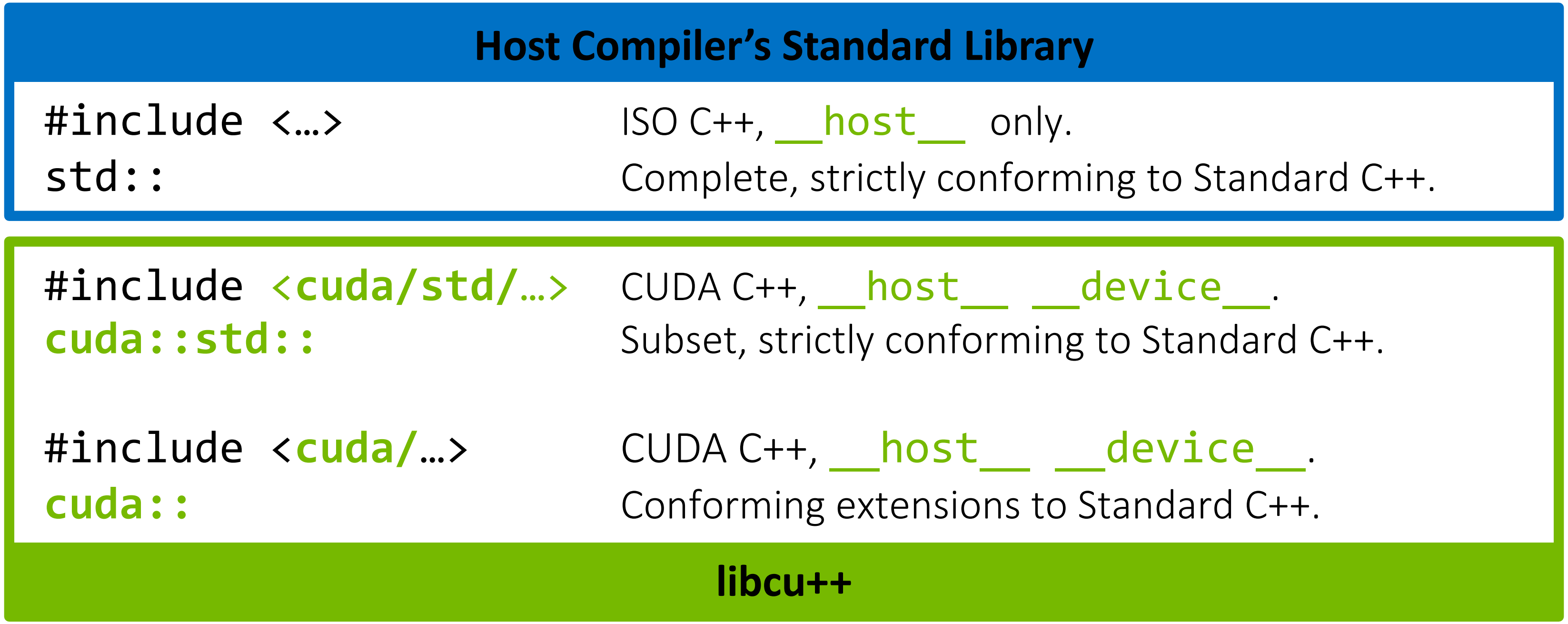
```
std::span A{input,  N * M};
std::span B{output, M * N};

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

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



mdspan

```
std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};

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

std::for_each(ex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[i, j] = A[i, j];
  });
```

LIBCU++: A GPU-ENABLED STL



Host Compiler's 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++.

libcu++

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

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, ...>`
…

C++ PARALLEL ALGORITHMS HINTS

No __device__ keyword needed for functions called in parallel algorithm, including lambdas.

Use heap memory, not stack.

- std::array<int, 1024> a = ...; //uses stack memory, will not work in device code.
- std::vector<int> v = ...; //uses heap memory, OK

For lambdas, capture by value, not reference; especially scalars.

Use Random Access Iterators.

- std::ranges::views::iota – When possible
- thrust::counting_iterator – When iota is not an option

No exceptions in GPU code.

STANDARD PARALLELISM RESOURCES

NVIDIA Developer Blogs

- Developing Accelerated Code with Standard Language Parallelism
- Accelerating Standard C++ with GPUs
- Accelerating Fortran DO CONCURRENT
- Bringing Tensor Cores to Standard Fortran
- Accelerating Python on GPUs with NVC++ and Cython

Legate and cuNumeric Resources

- https://github.com/nv-legate

Open-source codes

- LULESH - https://github.com/LLNL/LULESH
- STLBM - https://gitlab.com/unigehpfs/stlbm
- MiniWeather - https://github.com/mrnorman/miniWeather/
- POT3D - https://github.com/predsci/POT3D

C++ algorithms and execution policy reference

- https://en.cppreference.com/w/cpp/algorithm

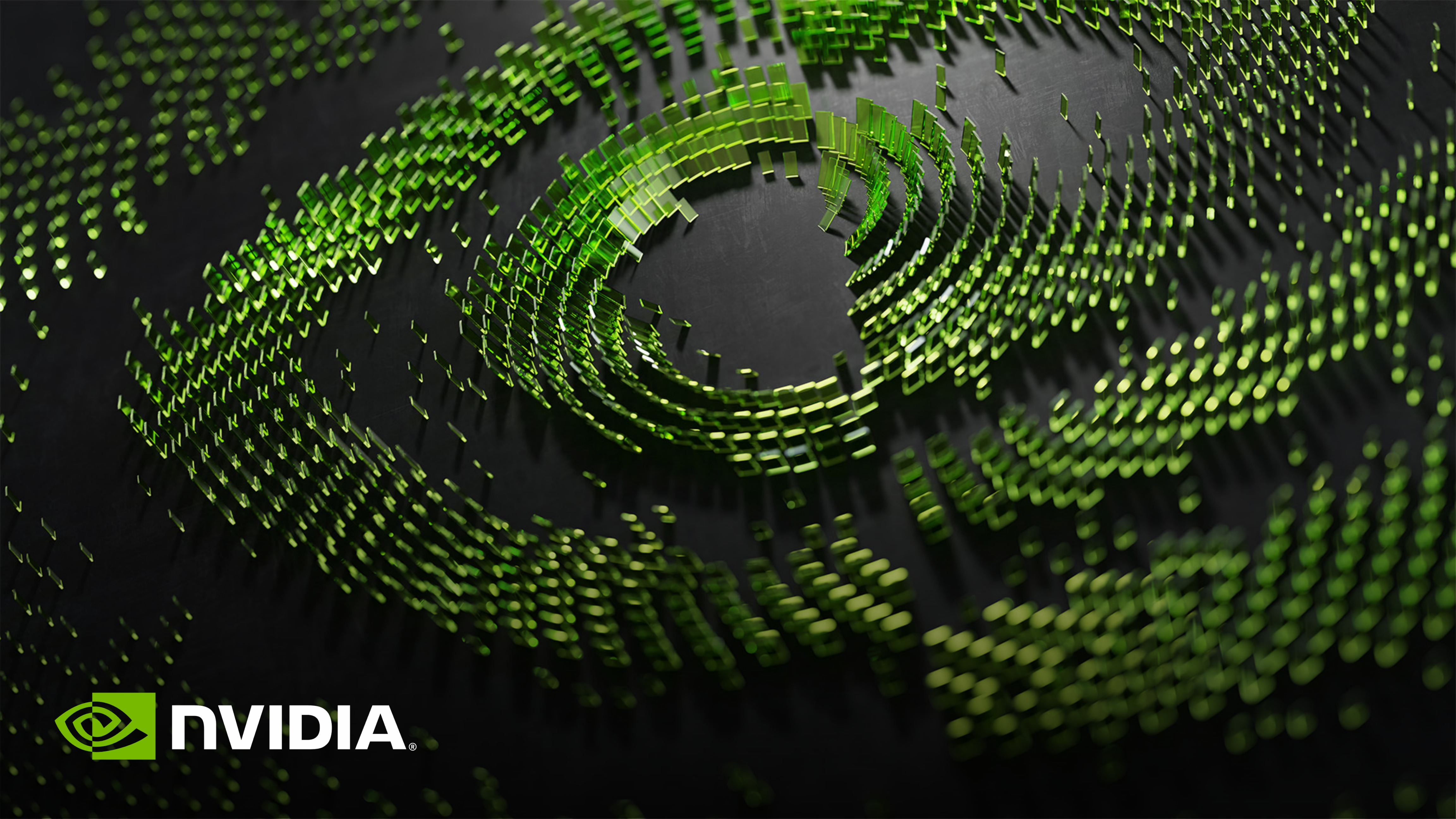
NVIDIA HPC Compilers Forum

- https://forums.developer.nvidia.com/c/accelerated-computing/hpc-compilers

Relevant GTC Spring 2022 Sessions

For more information on these topics

- No More Porting: Coding for GPUs with Standard C++, Fortran, and Python [S41496]
- A Deep Dive into the Latest HPC Software [S41494]
- C++ Standard Parallelism [S41960]
- Future of Standard and CUDA C++ [S41961]
- Shifting through the Gears of GPU Programming: Understanding Performance and Portability Trade-offs [S41620]
- From Directives to DO CONCURRENT: A Case Study in Standard Parallelism [S41318]



NVIDIA®