Portable Acceleration of HPC Applications with Standard C++

Gonzalo Brito Gadeschi, Jonas Latt, 2022

Who we are

Gonzalo Brito Gadeschi

- Senior Developer Technology Engineer Compute & HPC, NVIDIA Munich, Germany
- End-to-end HW/SW co-design
- Passionate about making accelerated computing more accessible
- Previously: CFD & multi-physics at RWTH Aachen University

Jonas Latt

- Associate Professor, Computer Science Department University of Geneva, Switzerland
- Research: HPC and CFD with applications to geophysical, biomedical and aero-spatial fields
- Palabos: open-source framework for lattice Boltzmann simulations of complex flows.
- Previously: fluid-dynamics research at Tufts University (Boston, USA) and EPFL (Lausanne, Switzerland). Co-founder of CFD company FlowKit.

Summary

Audience: students, developers, researchers and practitioners interested in developing portable HPC applications using ISO C++

Pre-requisites: experience with C++11 (lambdas, STL algorithms), multi-threaded programming, and MPI.

Content:

- **Lab 1:** BLAS DAXPY (fundamentals, beginner)
- **Lab 2:** 2D heat equation (MPI/C++ HPC application, intermediate)
- **Lab 3:** Parallel tree construction (concurrency, advanced)

AGENDA: part I

Introduction

- Accessing the Jupyter Notebooks
- Fundamentals of ISO C++ Parallelism
- Indexing, Ranges & Views

Lab 1: BLAS DAXPY

NVC++ unified & heterogeneous compiler

Lab 2: 2D heat equation (Part I)

- Multi-dimensional iteration

Performance portability of ISO C++ in HPC practice

Future topics in C++ parallelism

AGENDA: part II

Introduction to C++ Concurrency Primitives

- Threads, Atomics, Barriers, Mutexes

Lab 2: 2D heat equation (Part II)

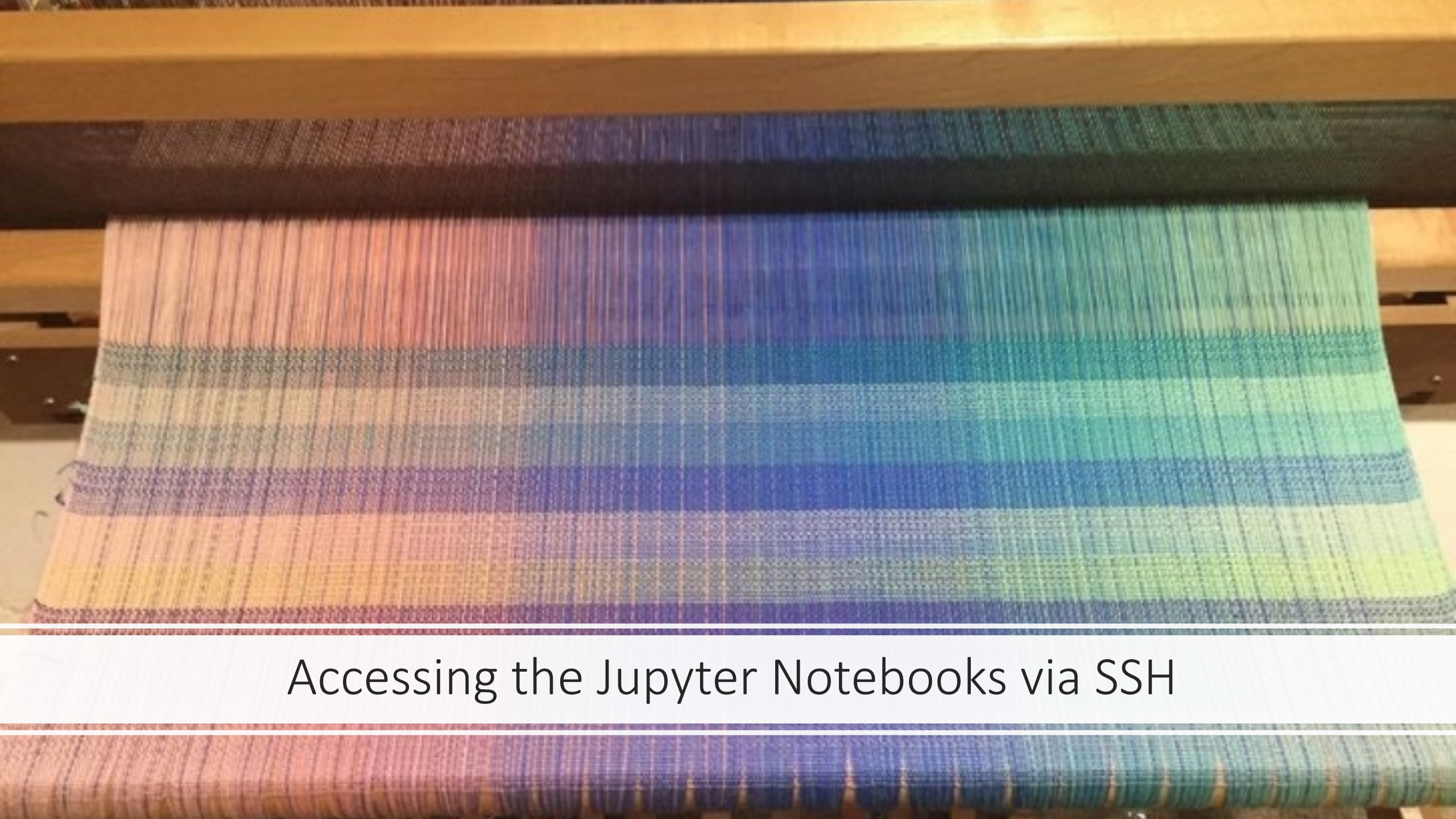
- Overlapping communication behind computation

Atomic memory operations

- Memory Orderings, Acquire/Release semantics
- C++ primitives: atomic, atomic_ref, atomic_flag, fences
- Critical sections & starvation-free algorithms

Execution Policies & Element Access Functions

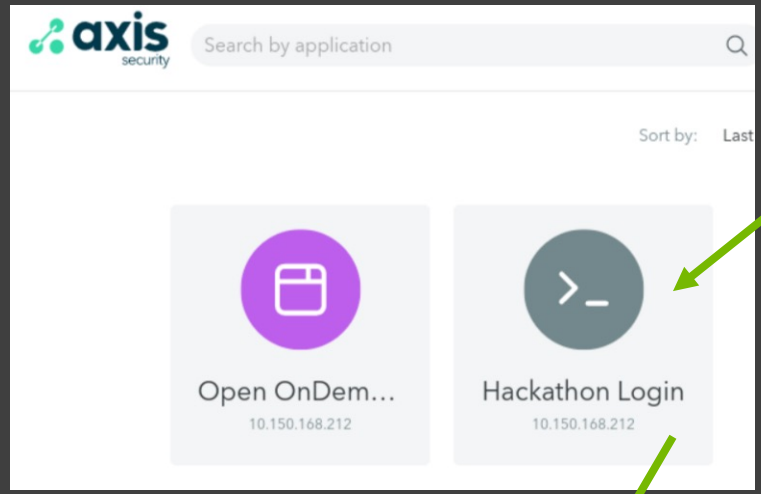
Lab 3: Parallel Tree Construction

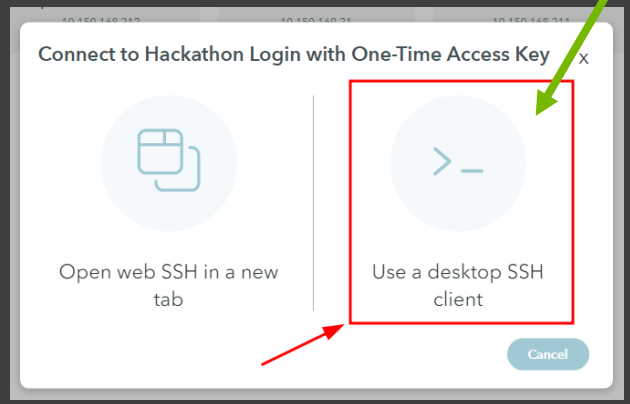


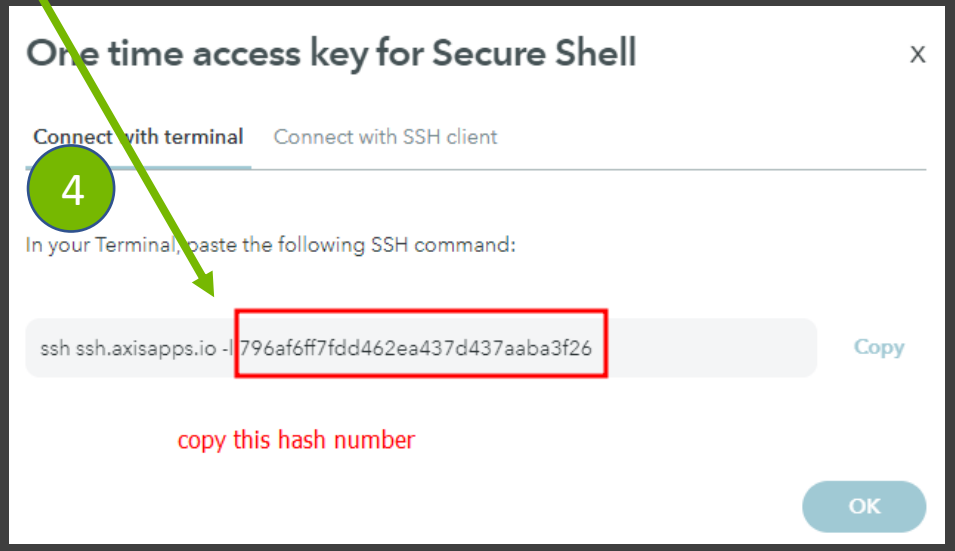
Accessing the Jupyter Notebooks via SSH

Log in to Open On Demand

1. Connect to the cluster via https://axis-raplabhackathon.axisportal.io/apps
2. Click on Hackathon Login
3. Open SSH on a desktop client
4. Copy the hash number

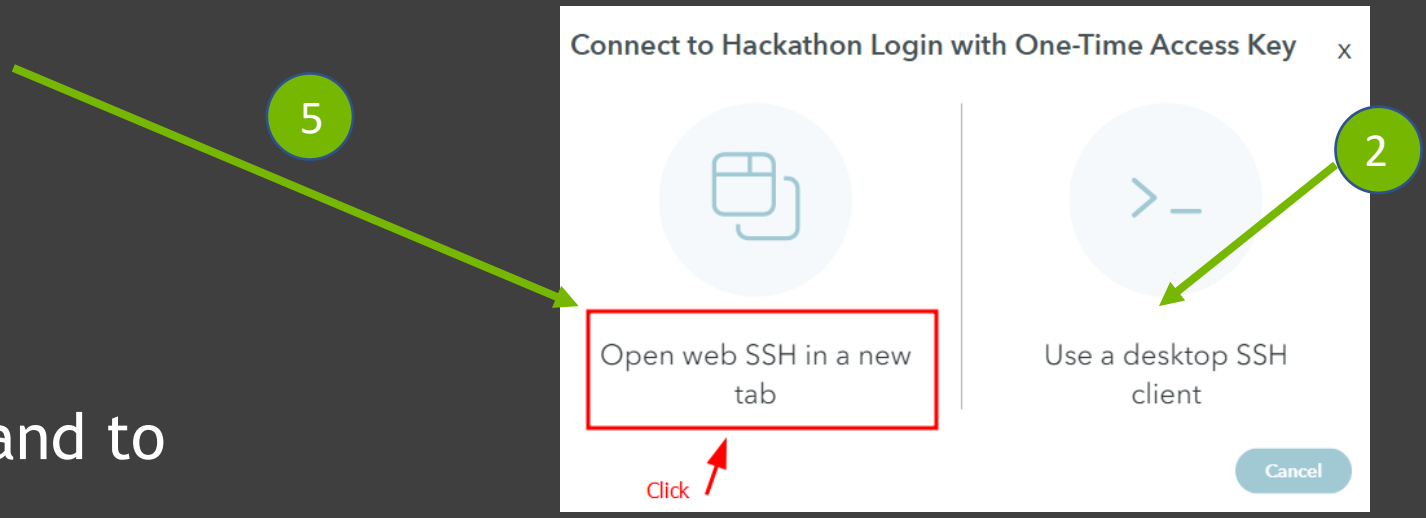






Start the Notebook job

5. Open SSH in a new Tab



6. Run the following command to launch the lab job:

```
sbatch /lustre/shared/bootcamps/stdpar_launch_script
```

7. Run this to check whether its running: `squeue -u $USER`

Connect to the notebook

5. The port_forwarding_command file contains:

```
$ cat port_forwarding_command
ssh –L localhost:8888:dgx05:8030 ssh.axisapps.io -l
```

6. Open a local terminal and copy the port forwarding command, adding the has from Step 4:

```
ssh -L localhost:8888:dgx05:8030 ssh.axisapps.io -l 796aff…
```

7. Open https://localhost:8888 in the browser; or with http (if https does not work)

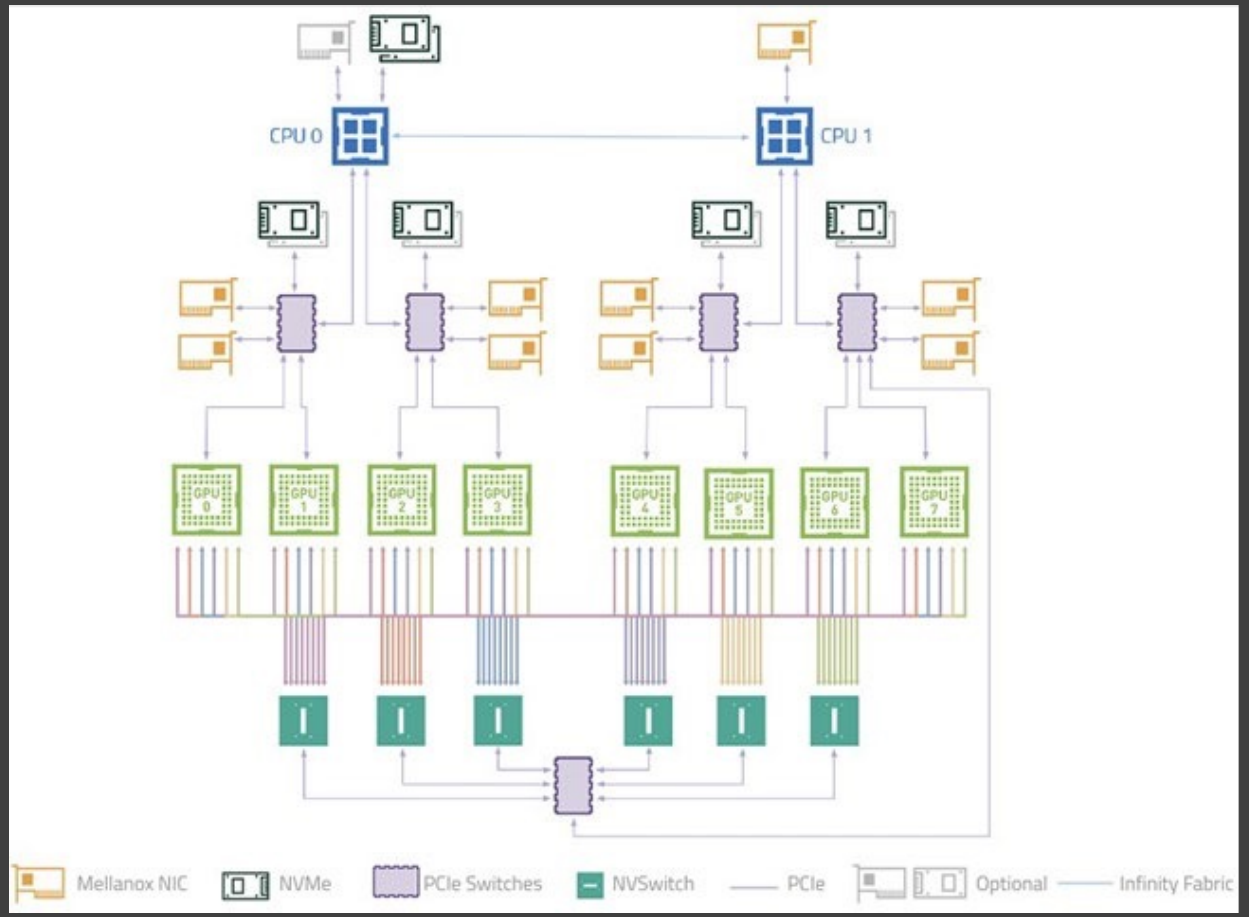
Cluster nodes: DGX-A100

2 CPU sockets

8 GPUs

Each GPU is physically partitioned with MIG into multiple GPUs

Each user gets its own MIG partition



Save your work!

If you want to preserve your modified exercises, these are stored to:

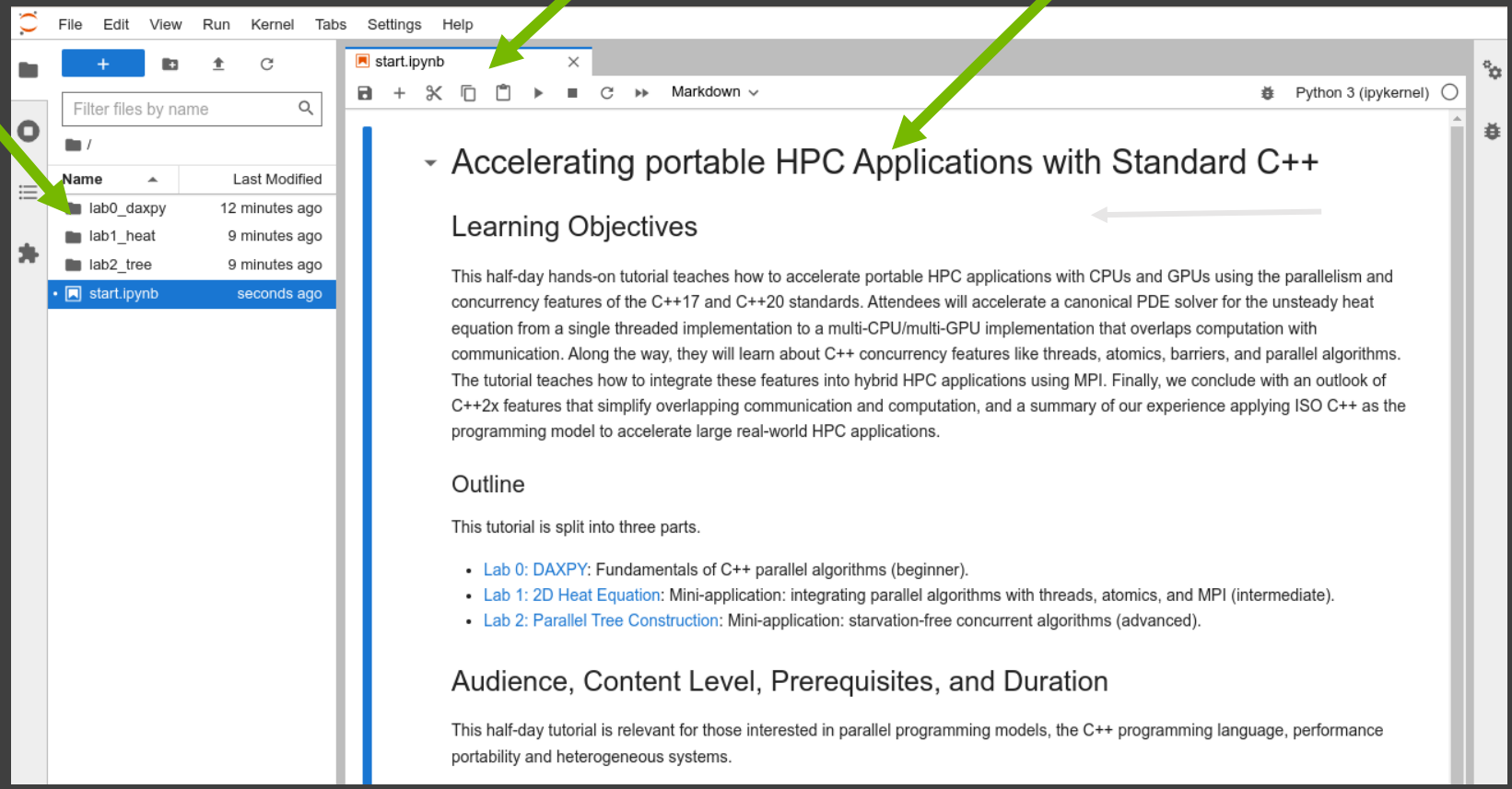
/lustre/workspaces/$USER/workspace-stdpar

Copy them after them course to your local machine using scp!

Using the Jupyer Notebook

File Explorer:

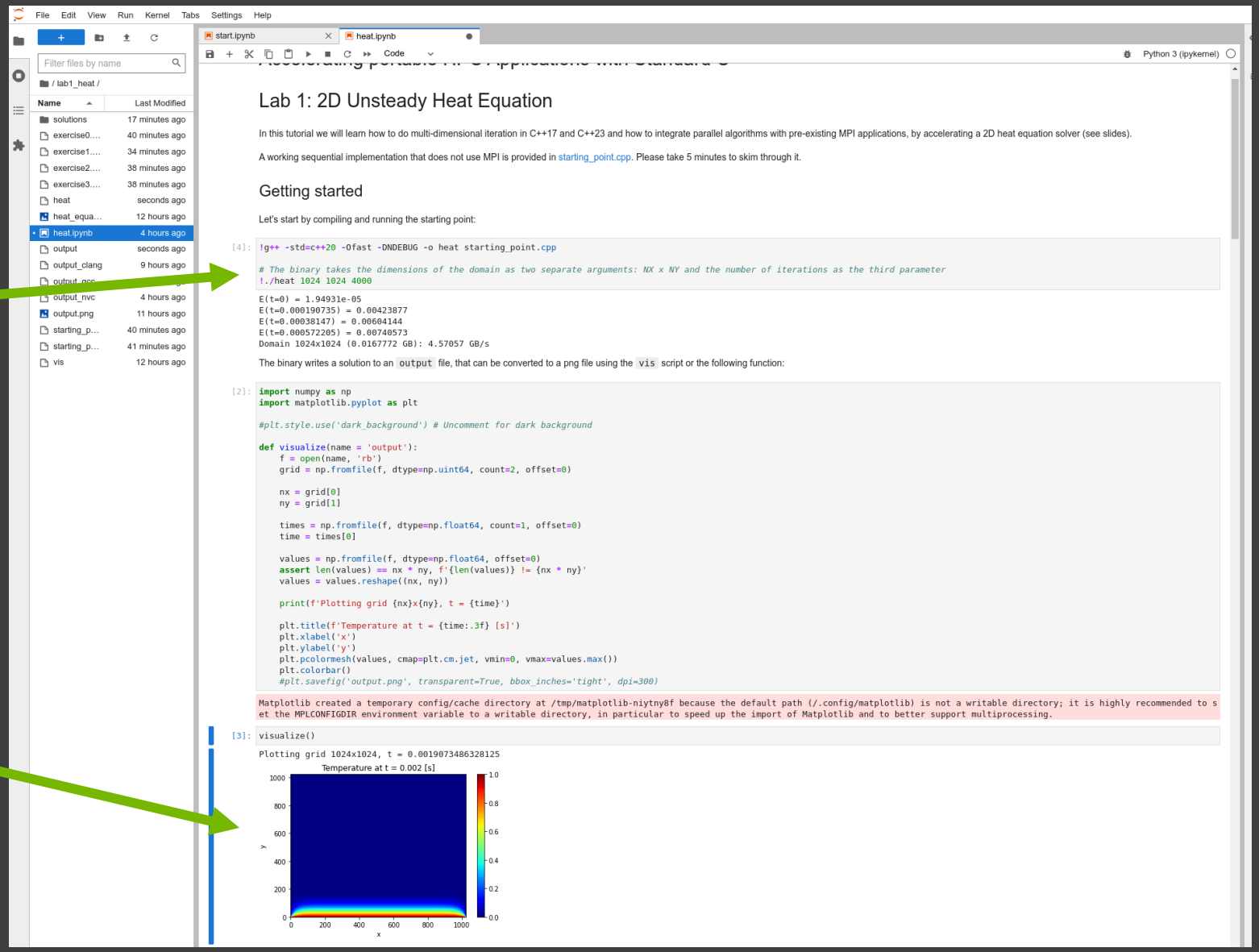
- Create
- Open
- Navigate



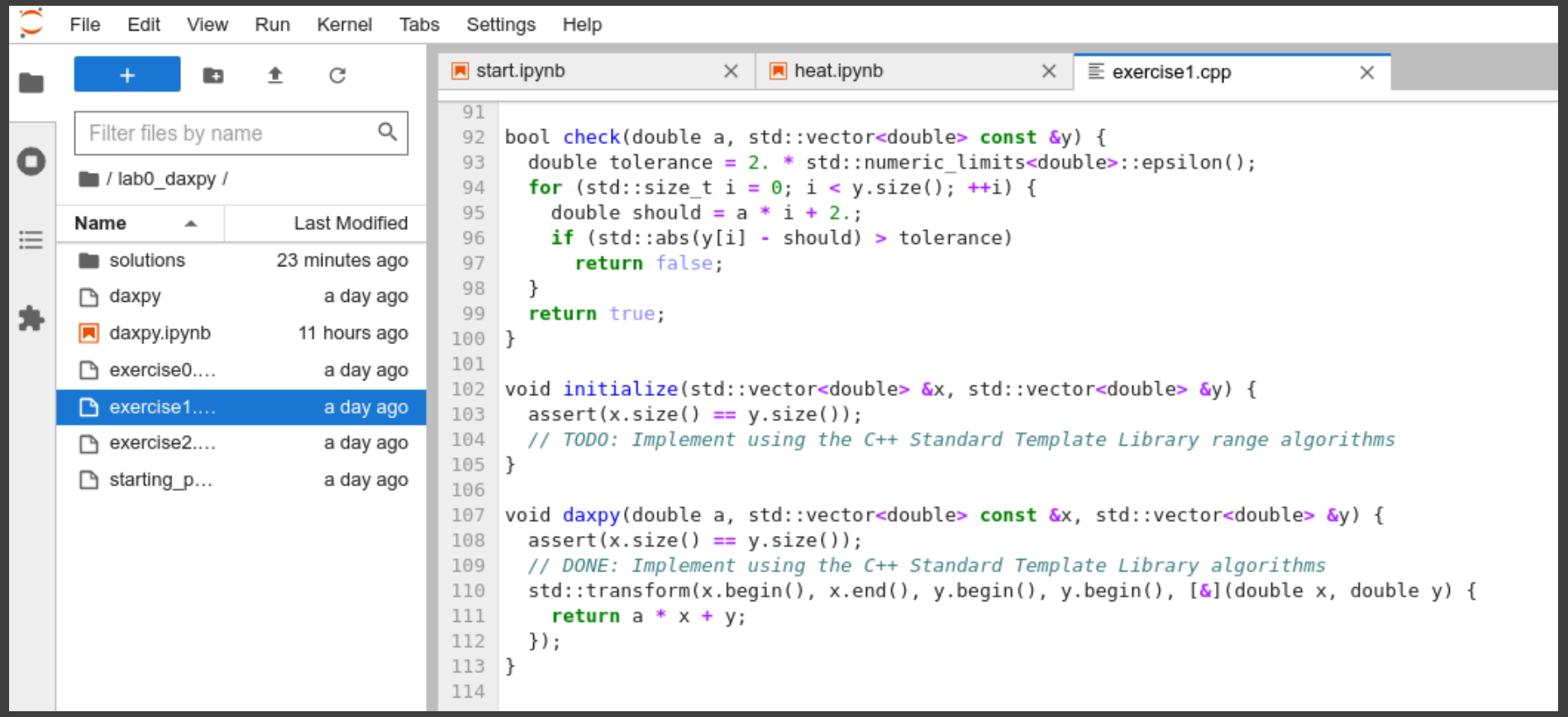
Labs

Run commands in code cells with CTRL+ENTER

Visualize the outputs of the simulations while learning

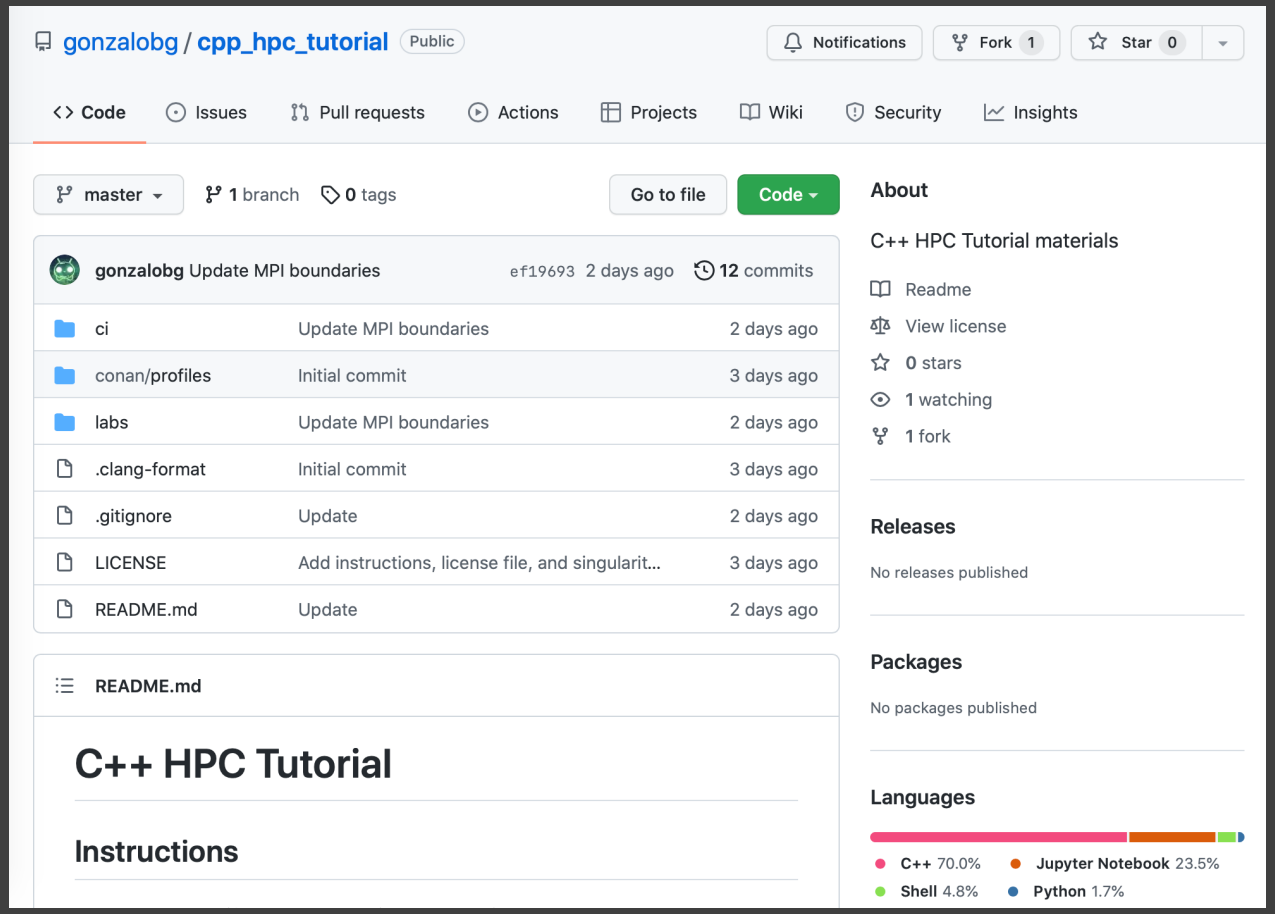


Labs: edit code in the browser



Labs and Notebook available on Github https://github.com/gonzalobg/cpp_hpc_tutorial

Follow the instructions to run them locally on your own hardware if desired.



C++ Pre-requisites

ISO C++ lambdas

Lambdas simplify the creation of function objects. This…

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [s,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);
```

ISO C++ lambdas

Lambdas simplify the creation of function objects. This…

…is equivalent to…

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [s,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

The [...] is called the "lambda capture" and controls how variables are stored within the lambda object:

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [s,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

The [...] is called the "lambda capture" and controls how variables are stored within the lambda object:

- [s] captures **s** "by value" (makes a copy)

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [s,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

The [...] is called the "lambda capture" and controls how variables are stored within the lambda object:

- [s] captures **s** "by value" (makes a copy)
- [&v] captures **v** "by reference" (stores a pointer)

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [s,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

Lambda captures support "capture defaults" that capture all variables used within the lambda:

- [&,s] captures s "by value" (makes a copy) and all other used variables "by reference" (store pointers)
- [=,&v] captures v "by reference" (stores a pointer to v) and all other used variables "by value" (copy them)

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [&,s](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

Lambda captures support "capture defaults" that capture all variables used within the lambda:

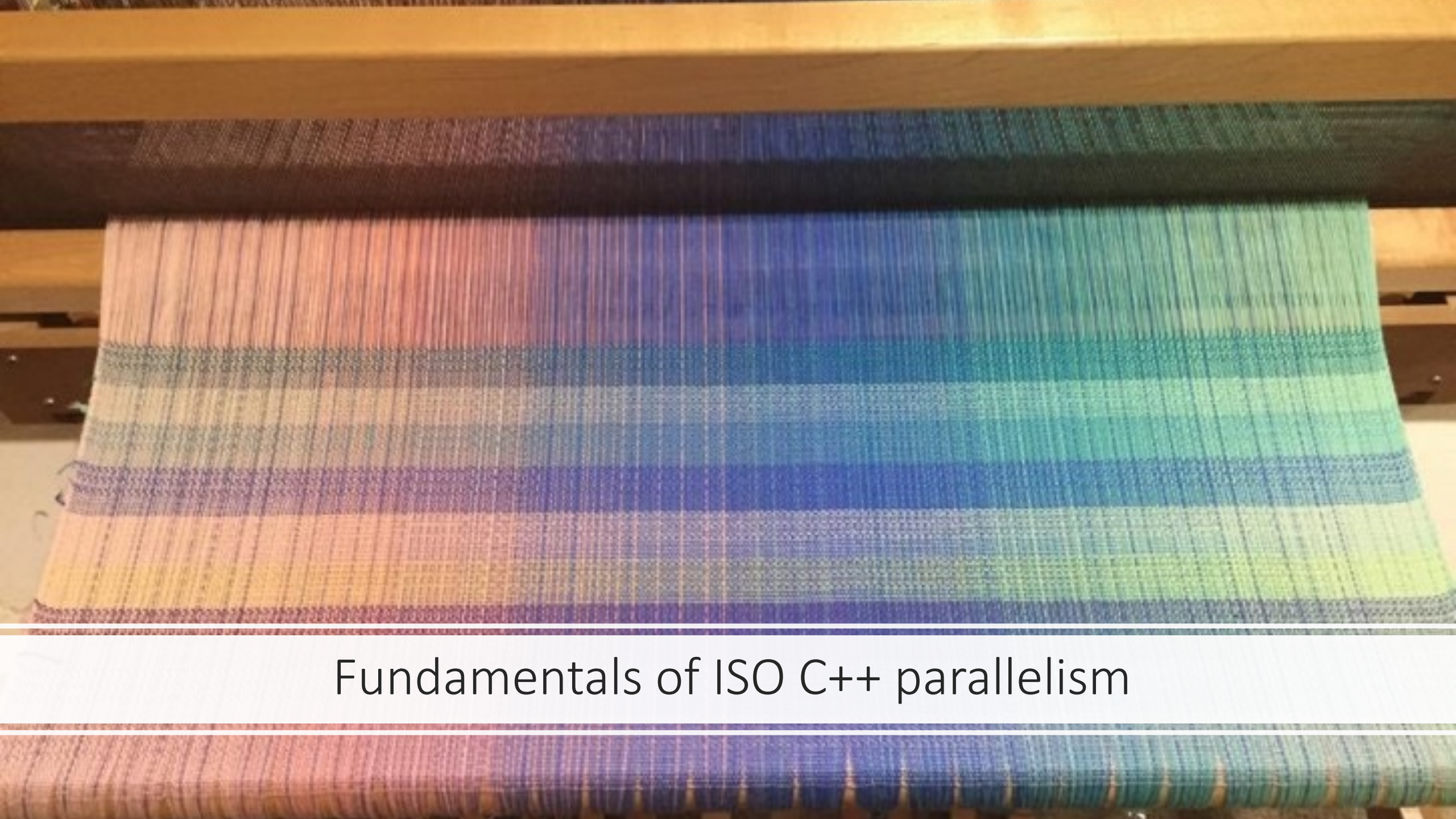
- [&,s] captures s "by value" (makes a copy) and all other used variables "by reference" (store pointers)
- [=,&v] captures v "by reference" (stores a pointer to v) and all other used variables "by value" (copy them)

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [=,&v](int idx) { return v[idx] * s; };
assert(f(1) == 4);

struct __unnamed {
    double s;
    std::vector<double>& v;
    double operator()(int idx) {
        return v[idx] * s;
    }
};
__unnamed f{s, v};
assert(f(1) == 4);
```

ISO C++ lambdas

Lambda captures support creating and assigning to new variables for use within the lambda:

```
std::vector<double> v = {1, 2, 3, 4};
double s = 2.;
auto f = [a = s, x = v.data()](int idx) {
    return x[idx] * a;
};
assert(f(1) == 4);
```



Fundamentals of ISO C++ parallelism

ISO C++ algorithms

In C++, containers can be processed by `for` loops...

```
std::vector<double> v = {1, 2, 3, 4}, w(4);
for (int i = 0; i < 4 ; ++i) {
    w[i] = 2. * v[i];
}
```

ISO C++ algorithms

In C++, containers can be processed by for loops...

```
std::vector<double> v = {1, 2, 3, 4}, w(4);
for (int i = 0; i < 4 ; ++i) {
    w[i] = 2. * v[i];
}
```

... or with standard template library (STL) algorithms, which are often more succinct.

```
std::transform(begin(v), end(v), begin(w),
    [](const double& el) {
        return 2. * el;
});
```

ISO C++ parallel algorithms Programming model introduced in C++17

```
std::vector<double> v = {1, 2, 3, 4}, w(4);
std::transform(std::execution::par, begin(v), end(v), begin(w),
               [](const double& el) { return 2. * el; });
```

ISO C++ parallel algorithms Programming model introduced in C++17

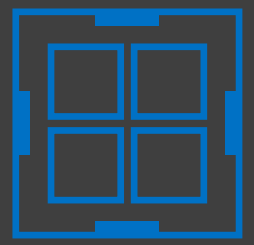
```
std::vector<double> v = {1, 2, 3, 4}, w(4);
std::transform(std::execution::par, begin(v), end(v), begin(w),
               [](const double& el) { return 2. * el; });
```

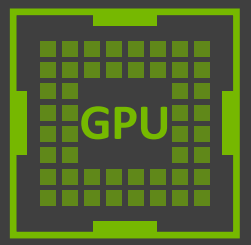
ISO C++ parallel algorithms
Compiler selects target for parallel execution

```
std::vector<double> v = {1, 2, 3, 4}, w(4);
std::transform(std::execution::par, begin(v), end(v), begin(w),
               [](const double& el) { return 2. * el; });
```

nvc++ -stdpar=multicore

nvc++ -stdpar=gpu





ISO C++ parallel algorithms
Hybrid (CPU / GPU) program execution

```
std::vector<double> v = {1, 2, 3, 4}, w(4);

// Data is first processed sequentially on the host (CPU)
std::transform(begin(v), end(v), begin(w),
               [](const double& el) { return 2. * el; });

// Then, the same data is processed in parallel, e.g. on a GPU
std::transform(std::execution::par, begin(w), end(w), begin(w),
               [](const double& el) { return 2. * el; });
```

- Same data can be accessed from the CPU and from the GPU.
- Memory transfer is implicit.
- Use of a managed memory model.

ISO C++ parallel algorithms Accelerator support limitation

Stack variable “a” is captured by reference (&). Accelerators read it remotely from the CPU thread stack.

- Non-coherent HW (PCIe): not supported
- Coherent HW (Grace Hopper): poor performance.
- Note: this is a problem for stack data only, not for heap data.

```
void multiply_with(vector<double>& v, double a) {
  std::for_each(std::execution::par,
                begin(v), end(v),
                [&](double& x) { x *= a; }
  );
}
```

ISO C++ parallel algorithms Accelerator support limitation

Stack variable “a” is captured by reference (&). This is problematic (non-supported or slow).

```
void multiply_with(vector<double>& v, double a) {
  std::for_each(std::execution::par,
                begin(v), end(v),
                [&](double& x) { x *= a; }
  );
}
```

Solution: Stack variable “a” is captured by value (=) and copied to the accelerator.

```
void multiply_with(vector<double>& v, double a) {
  std::for_each(std::execution::par,
                begin(v), end(v),
                [=](double& x) { x *= a; }
  );
}
```

ISO C++ parallel algorithms References

CppCon talks:

- Thomas Rodgers, *Bringing C++ 17 Parallel Algorithms to a standard library near you*, 2018
- Olivier Giroux, *Designing (New) C++ Hardware*, 2017
- Dietmar Kühl, *C++17 Parallel Algorithms*, 2017
- Bryce Adelstein Lelbach, *The C++17 Parallel Algorithms Library and Beyond*, 2016

GTC talks:

- Simon McIntosh-Smith et al., *How to Develop Performance Portable Codes using the Latest Parallel Programming Standards*, Spring 2022
- Jonas Latt, *Porting a Scientific Application to GPU Using C++ Standard Parallelism*, Fall 2021
- Jonas Latt, *Fluid Dynamics on GPUs with C++ Parallel Algorithms: State-of-the-Art Performance through a Hardware-Agnostic Approach*, Spring 2021

C++ Parallel Algorithms in C++17 & C++20

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

Iteration & Transform

std::for_each, std::for_each_n
std::transform, std::transform_reduce
std::transform_inclusive_scan, std::transform_exclusive_sca

Reductions

std::reduce, std::transform_reduce
std::exclusive_scan, std::inclusive_scan
std::adjacent_difference
std::all_of, std::any_of, std::none_of
std::count, std::count_if
std::is_sorted, std::is_sorted_until, std::is_partitioned
std::is_heap, std::is_heap_until
std::max_element, std::min_element, std::minmax_element
std::equal, std::lexicographical_compare

Searching

std::find, std::find_if, std::find_if_not, std::find_end, std::find_first_of
std::adjacent_find, std::mismatch
std::search, std::search_n

Memory movement & Initialization

std::copy / copy_if / copy_n / move / uninitialized_...
std::fill, std::fill_n, std::uninitialized_...
std::generate, std::generate_n
std::swap_ranges
std::reverse, std::reverse_copy

Removing & replacing elements

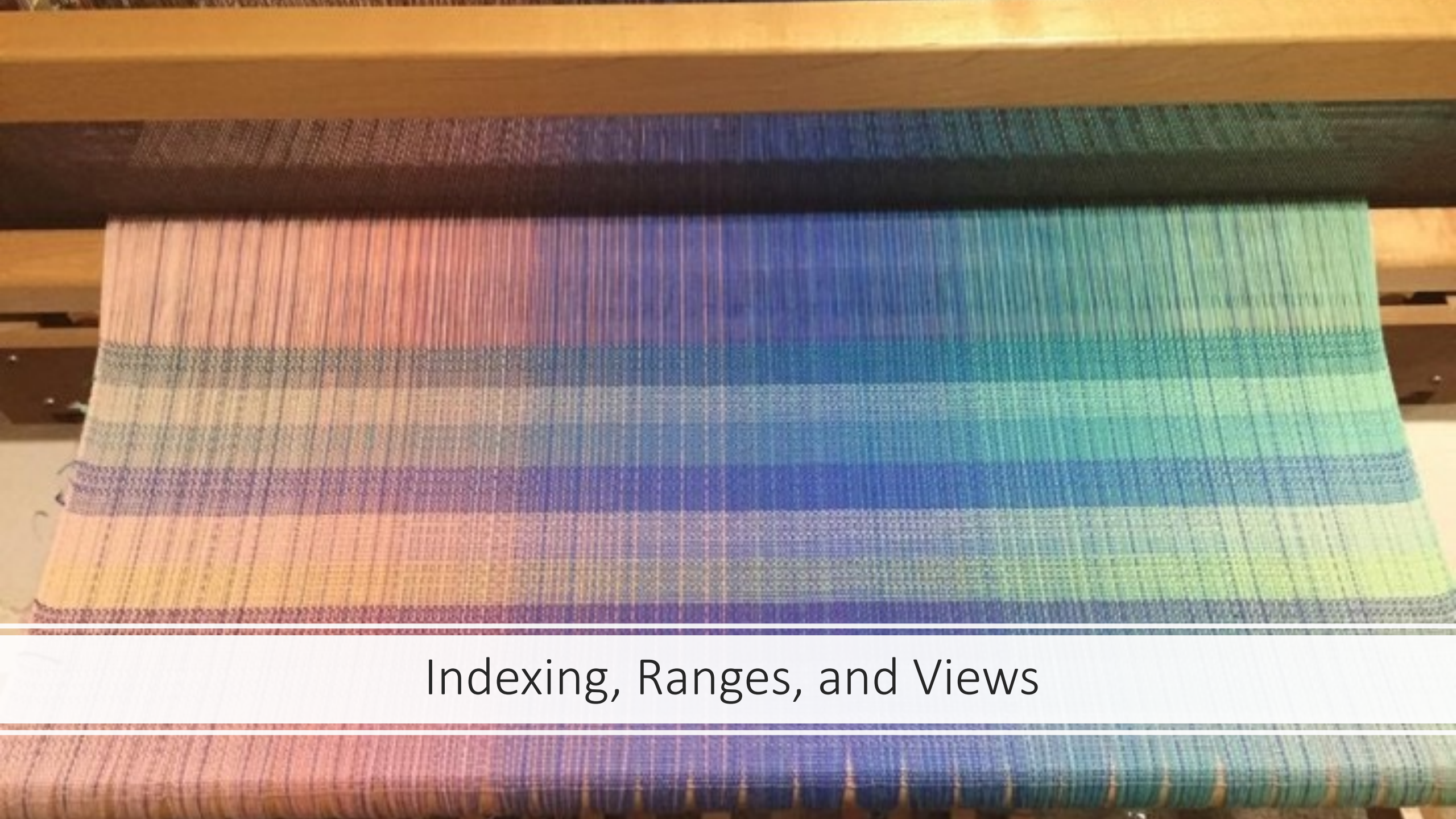
std::remove, std::remove_if
std::replace, std::replace_if, std::replace_copy, std::replace_copy_if
std::unique / std::unique_copy

Reordering elements

std::sort, std::stable_sort, std::partial_sort, std::partial_sort_copy
std::rotate, std::rotate_copy, std::shift_left, std::shift_right
std::partition, std::partition_copy, std::stable_partition
std::nth_element
std::merge, std::inplace_merge

Set operations

std::includes, std::set_intersection, std::set_union
std::set_difference, std::set_symmetric_difference



Indexing, Ranges, and Views

How to find the index of an element?

With C++ for loops we have the index...

```
std::vector<double> v = {1, 2, 3, 4};
for (int i = 0; i < 4 ; ++i) {
    v[i] = f(i);
}
```

...with parallel algorithms we do not...

```
std::transform(begin(v), end(v),
    [](const double& el) {
        return f(???);
});
```

How to find the index of an element? Option 1: obtain index from address

With C++ for loops we have the index...

```
std::vector<double> v = {1, 2, 3, 4};
for (int i = 0; i < 4 ; ++i) {
    w[i] = f(i);
}
```

...capture pointer to data by value (=) and compute the index from the memory address of the element...

```
std::transform(begin(v), end(v),
    [v = v.data()](const double& el) {
        ptrdiff_t i = &el - v;
        return f(i);
});
```

How to find the index of an element? Option 2: use a counting iterator

A counting iterator is an iterator that wraps an index:

- boost::counting_iterator
- thrust::counting_iterator

```
thrust::counting_iterator<size_t> it{0};
assert(*it == 0);
++it;
assert(*it == 1);
```

… capture a pointer to the data by value (=) and use a counting iterator with the std::for_each_n algorithm…

```
std::for_each_n(it, v.size(),
    [v = v.data()](size_t i) {
        v[i] = f(i);
});
```

How to find the index of an element? Option 3: use C++20 Ranges and Views

The function *iota* from the C++20 collection of views defines an iterable sequence of numbers without actually allocating them.

Similarly, you can iterate over n-dimensional array indices using the view *cartesian_product*.

```
auto ints = std::views::iota(0, 4);
std::for_each(par, begin(ints), end(ints),
    [v = v.data()](size_t i) {
        v[i] = f(i);
});

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

std::for_each(par, begin(v), end(v),
    [](auto& e) {
        auto [i, j] = e;
});
```

How to find the index of an element? Summary

- Use pointer arithmetic (C++17)
  - In the lambda argument, pass the element by reference
  - Retrieve the index from the address of the element
- Use counting_iterator from a library (C++17)
  - Available in Thrust
  - Available in Boost
  - Available in other header files found on GitHub
- Use C++20 views
  - C++20 view *iota* for 1-D indexing. E.g. views::iota(0, N).begin()
  - C++23 view *cartesian_product* for n-D indexing.

Background: C++20 Ranges and Views

A Range is an object that provides a pair of iterators denoting a range of elements, a std::vector is a range.

```
std::vector<double> v = {0, 1, 2, 3};
auto b = v.begin();
auto e = v.end();
```

Sequential version of the C++ STL algorithms have versions that accept Ranges…

Parallel versions of the algorithms do not!

```
// Iterator version:
std::transform(begin(v), end(v),
    [](const double& el) { return 2. * el; });

// Range version:
std::ranges::transform(v,
    [](const double& el) { return 2. * el; });
```

Background: C++20 Ranges and Views

Views are lazy Range algorithms that produce elements as iterated over...

```
auto ints = std::views::iota(0, 4);
for (int i : ints) {
    v[i] = f(i);
}
```

...we can use views::iota to generate a range of indices...

...that we can use with the parallel STL algorithms by using its iterators...

```
std::for_each(par, begin(ints), end(ints),
    [v = v.data()](size_t i) {
        v[i] = f(i);
});
```

Background: C++20 Ranges and Views

Views algorithms compose via the ”pipe” operator |…

```
auto seq =  views::iota(0, N)
          | views::filter(is_prime)
          | views::stride(2)
          | views::transform(square);
for (auto i : seq) cout << i << “,”;
// Prints: 4, 25, 121, ….
```

…Ranges and Views compose as well…

```
std::vector<int> v{0, 1, 2, 3};
for (int e : v | views::filter(even))
    cout << e << “,”;
// Prints 0, 2
```

C++20 and C++23 Views

- views::all
- views::filter
- views::transform
- views::take
- views::join
- views::split
- views::zip

- views::counted
- views::reverse
- views::keys
- views::values
- views::cartesian_product
- etc.

```
std::vector<int> xs{0, 1, 2, 3}, ys{4, 5, 6, 7};
for (auto [x, y] : views::zip(xs, ys))
    cout << “(“ << x << “,” << y << “), ”;
// Prints: (0,4), (1,5), (2,6), (3,7)
```

range-v3: https://github.com/ericniebler/range-v3
Many Views and more for C++14 onwards

```
std::vector<int> w{4, 5, 6, 7};
for (auto [x, y] : w | range::views::enumerate)
    cout << “(“ << x << “,” << y << “), ”;

// Prints: (0,4), (1,5), (2,6), (3,7)
```

References

- Tristan Brindle, An Overview of Standard Ranges, CppCon 2019
- Eric Niebler, Ranges for the Standard Library, CppCon 2015

LAB 1: BLAS DAXPY

BLAS DAXPY: Double-precision AX + Y

Allocate memory…

```
std::vector<int> x(N), y(N);
```

Initialize x and y…

```
for (int i = 0; i < N; ++i) {
    x[i] = …;
    y[i] = …;
}
```

Update y…

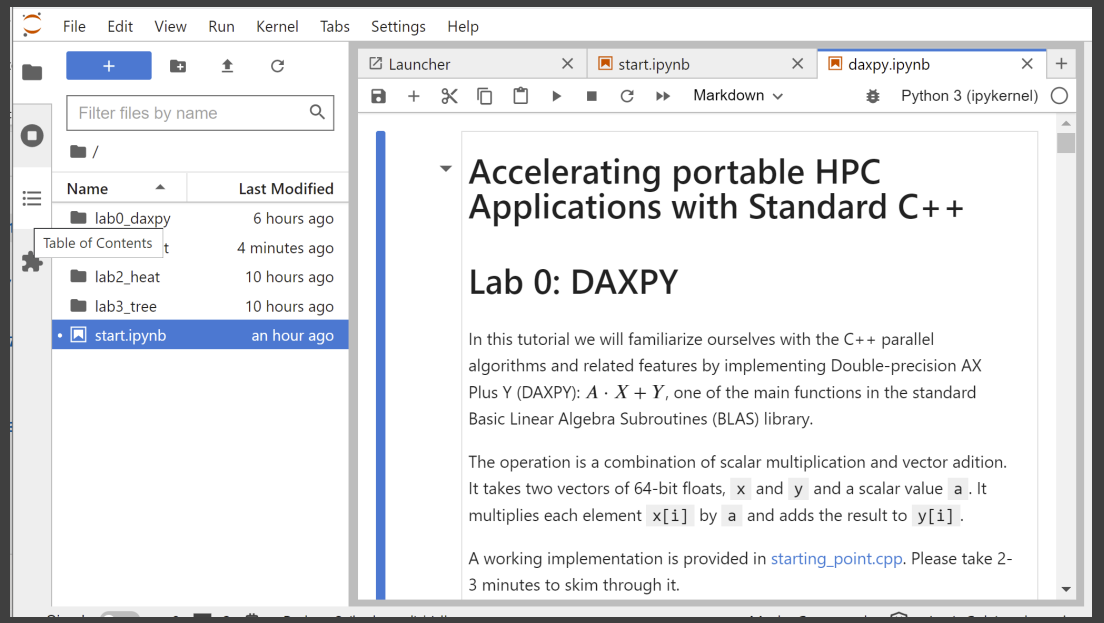
```
for (int i = 0; i < N; ++i) {
    y[i] += a * x[i];
}
```

Lab 1: BLAS DAXPY
3 Exercises

- **Exercise 1:** rewrite the for-loop implementation of the BLAS DAXPY kernel using sequential STL algorithms

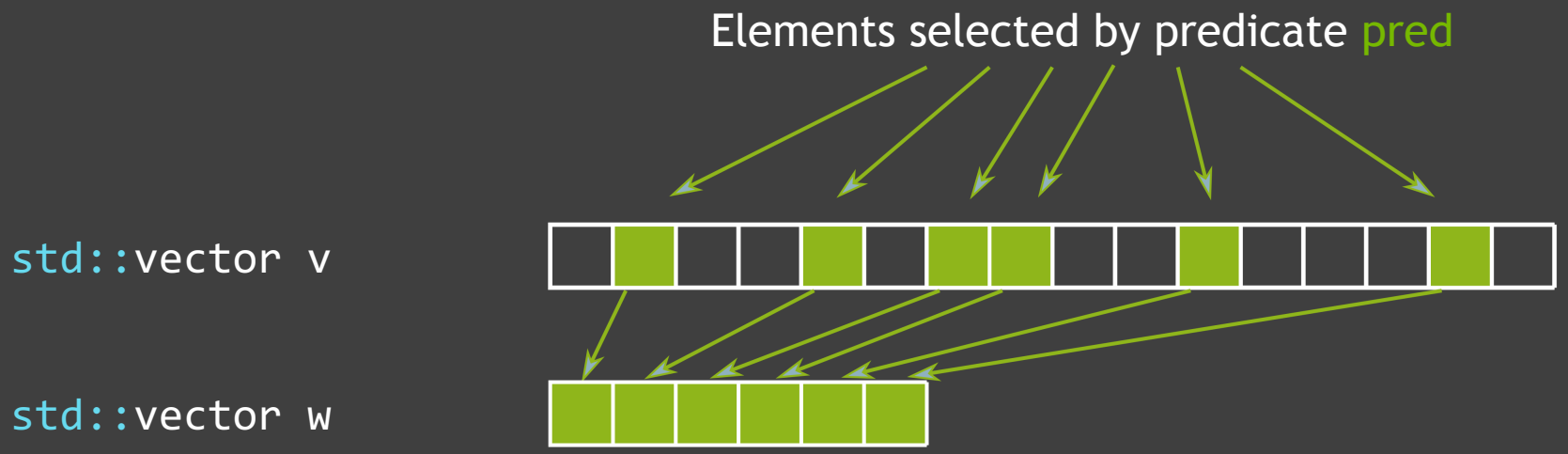
- **Exercise 2:** rewrite the for-loop implementation of the "initialization" kernel using sequential STL algorithms with any of the indexing approaches discussed in the tutorial

- **Exercise 3:** parallelize the STL versions of the application using the Execution Policies



Lab 1: Select

- **Exercise 1:** write the function select, which copies selected values from v to w.

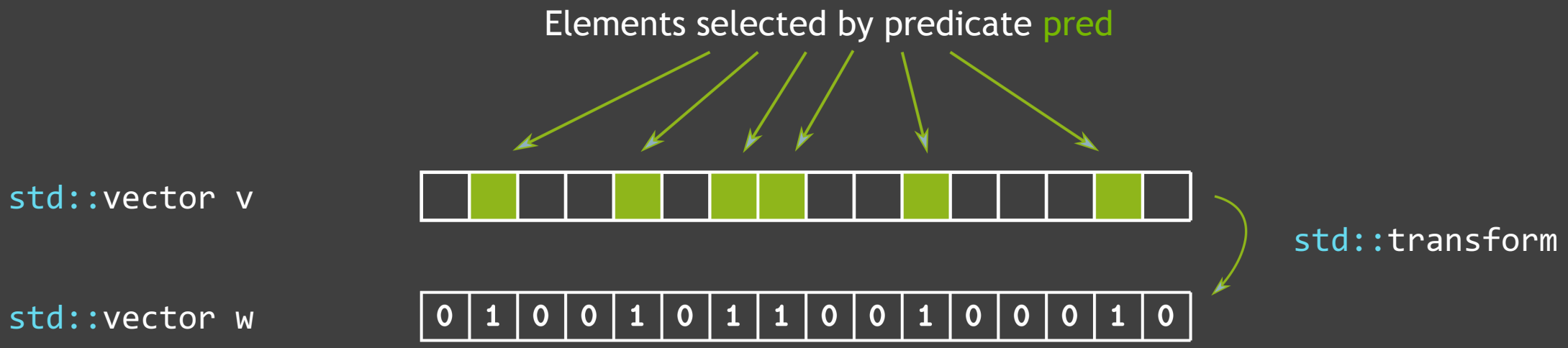


- Sequentially no problem, but how to write a parallel algorithm ?

```
template<class UnaryPredicate>
std::vector<int> select( const std::vector<int>& v,
                         UnaryPredicate pred );
```

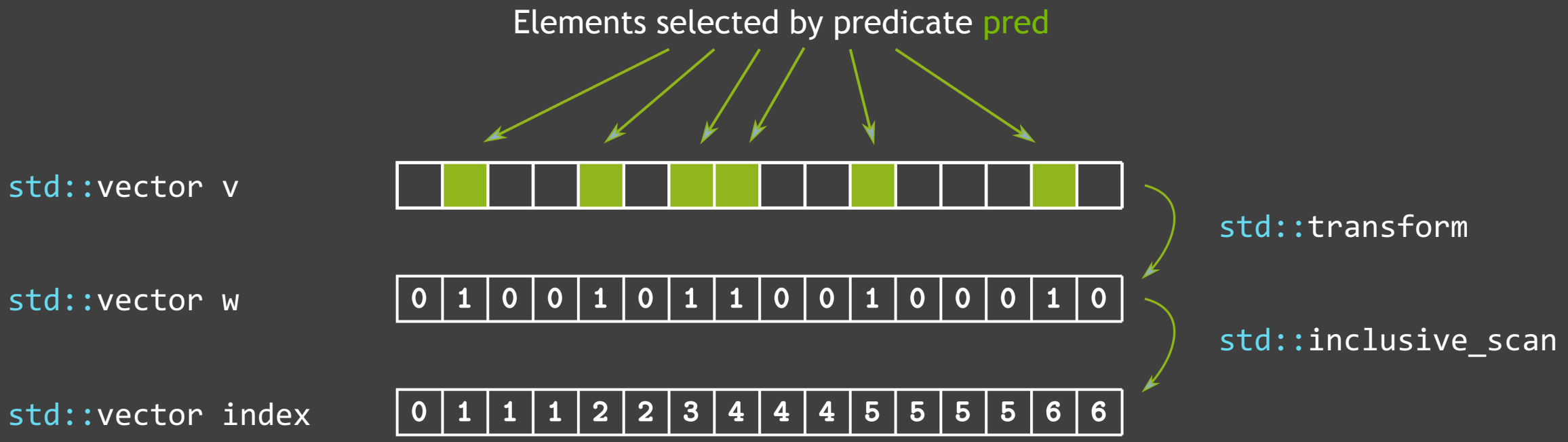
Lab 1: Select

- **Step 1:** Write out a binary-valued array for the values of the predicate.



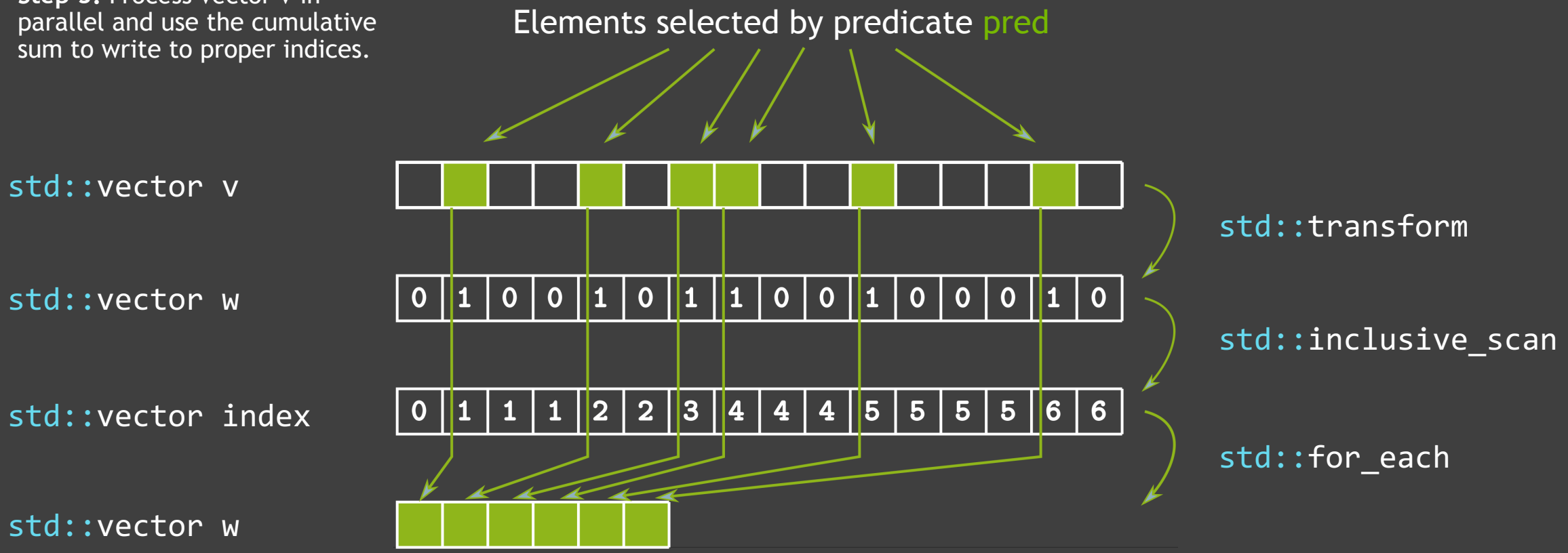
Lab 1: Select

- **Step 1:** Write out a binary-valued array for the values of the predicate.
- **Step 2:** Compute the cumulative sum of this array.



Lab 1: Select

- **Step 1:** Write out a binary-valued array for the values of the predicate.
- **Step 2:** Compute the cumulative sum of this array.
- **Step 3:** Process vector v in parallel and use the cumulative sum to write to proper indices.



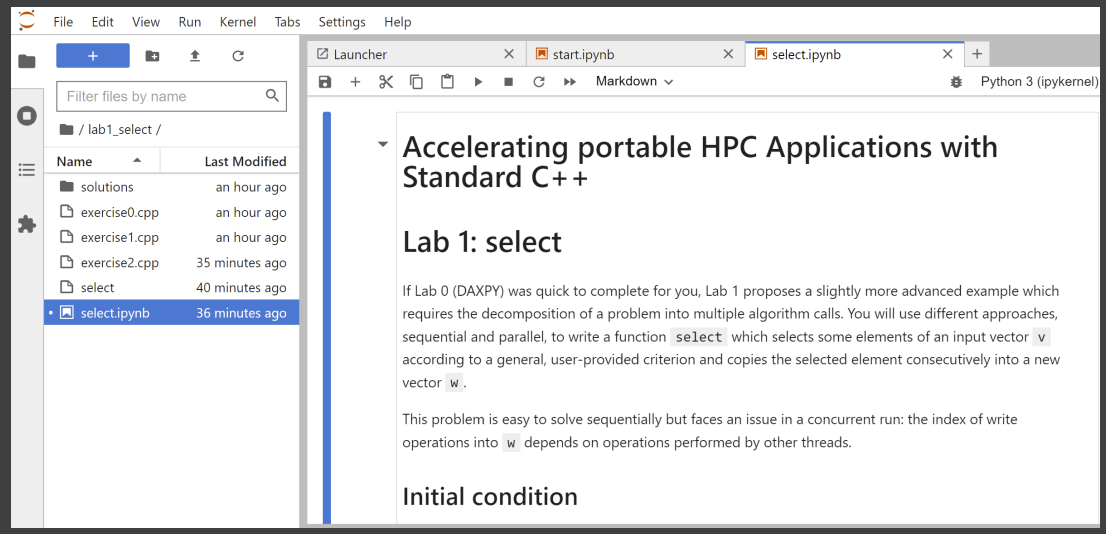
Lab 1: Select
3 Exercises

- **Lab 1** is optional and should be done only if you had the time to complete Lab 0.

- Exercise 1: write a sequential version of the function select that uses the algorithm std::copy_if and a back inserter for vector w.

- **Exercise 2:** write a parallel version of the function select that works with two temporary vectors v_sel and index.

- **Exercise 3:** reduce the number of steps from 2 to 3 and avoid the creation of v_sel using the algorithm transform_inclusive_scan.

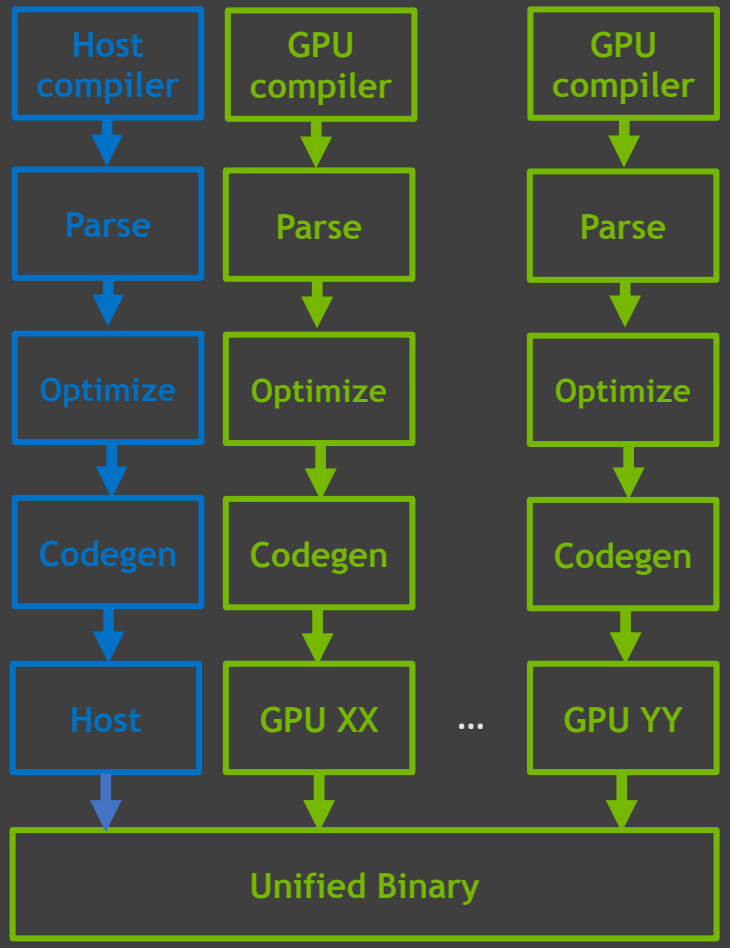


Lab 1: BLAS DAXPY Solutions (DEMO)

Lab 1: Select Solutions (DEMO)

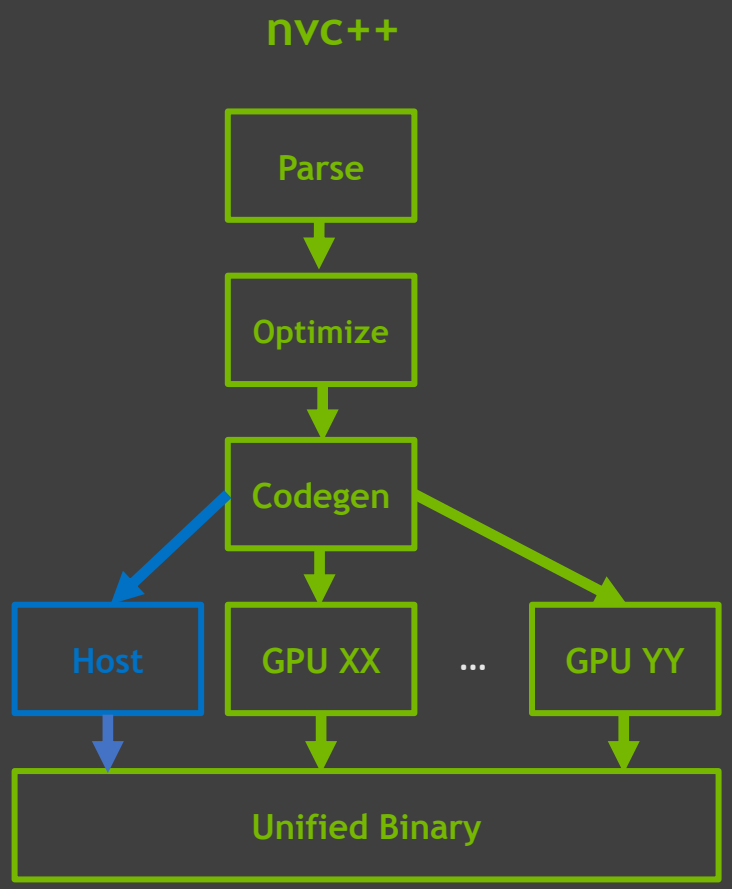
NVC++: A unified compiler for heterogeneous systems

NVCC: NVIDIA CUDA COMPILER
Separate compilation of host and device code



- Host and Device compilers don't know anything about each other.
- Allows user to provide their own host compiler!
- Application code is expanded multiple times for the different targets

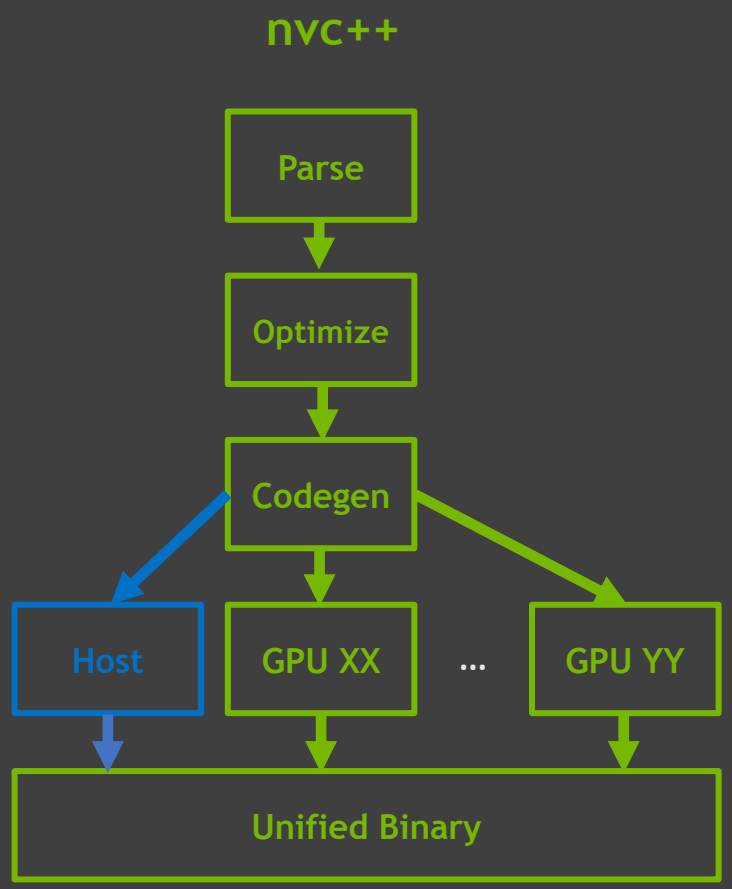
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- Execution space inference for code reachable from a translation unit:

```
// No __host__ __device__ annotations
int square(int x) { return x * x; }

// Host:
int y = square(2);

// Device:
std::transform(par, begin(x), end(x),
               [&](int& x) { return square(x); });
```

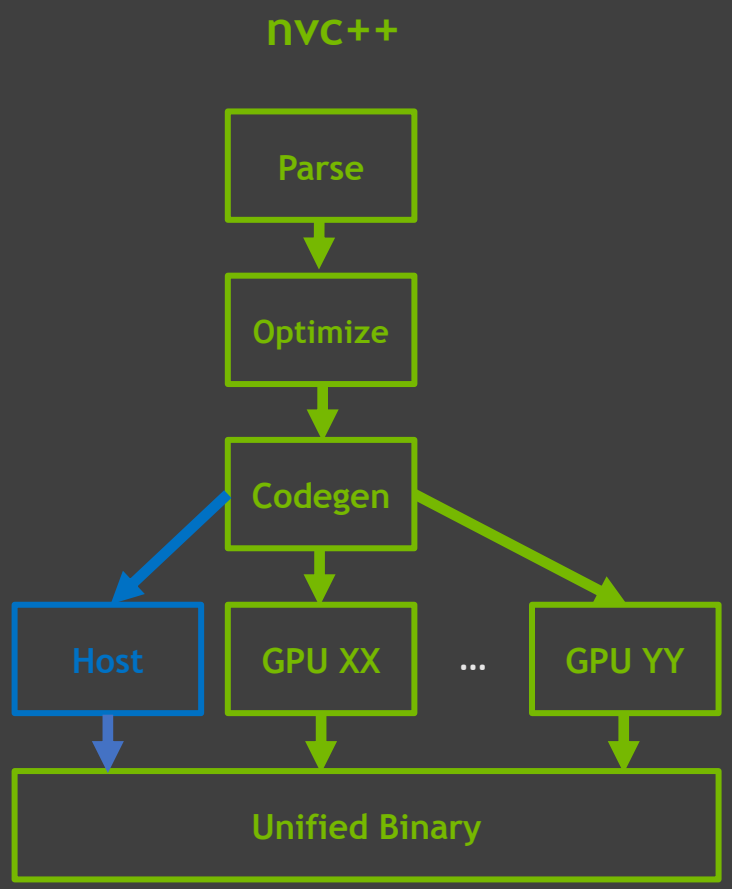
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- Mix ISO C++, OpenMP, OpenAcc, and CUDA:

```
#pragma omp loop …
for (int i = 0; i < N; ++i) { … }

#pragma acc kernels …
for (int i = 0; i < N; ++i) { … }

kernel<<<…>>>(…); // CUDA

#pragma omp target data map(tofrom:x[0:N])
{
    std::transform(par, begin(x), end(x),
                   [&](int& x) { return square(x); });
}
```

NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler

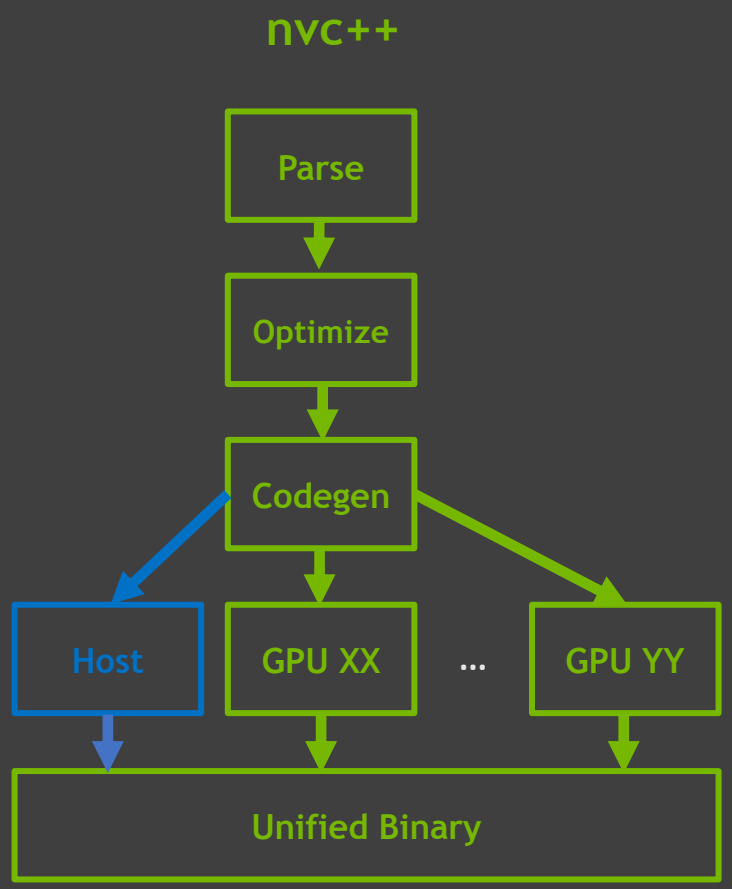


- Separately compile host and device code:

```
// File a.cpp
__host__ __device__
int square(int) { … }

#pragma omp declare target
int twice(int) { … }
```

```
// File b.cpp
std::transform(par, begin(x), end(x), [&](int& x) {
    return square(twice(x));
});
```

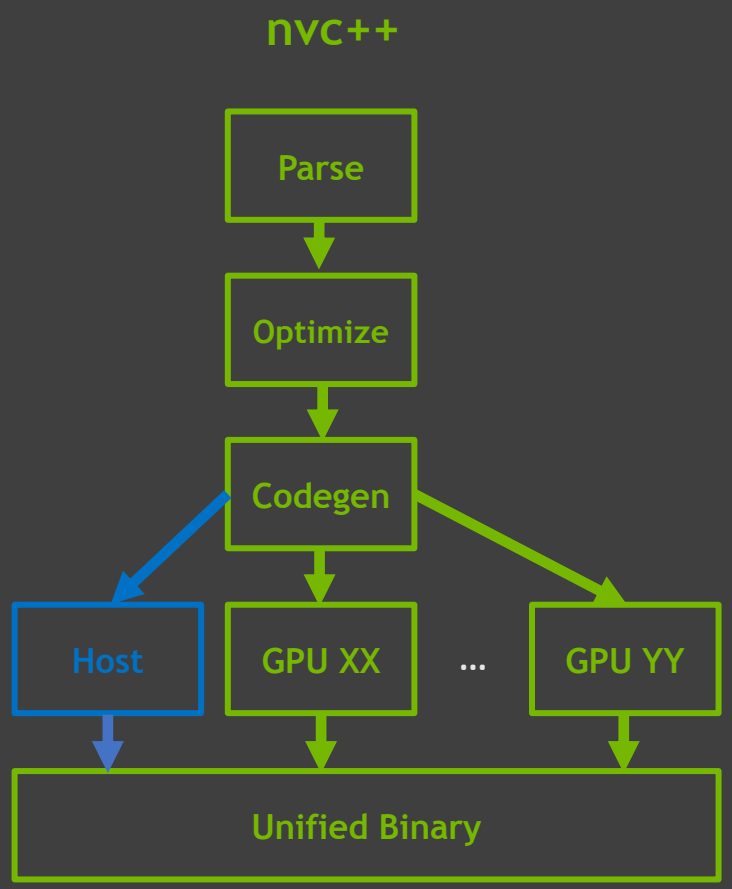
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- Generic lambdas

```
std::transform(par, begin(x), end(x), [&](auto x) {
    return square(twice(x));
});
```

NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- Heterogeneous C++ atomics

```
#include <atomic>
std::atomic<int> red = 0;

std::transform(par, begin(x), end(x), [&](int& x) {
    red.fetch_add(x);
    return square(twice(x));
});
```

NVIDIA C++ STANDARD LIBRARY
For your entire system

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

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

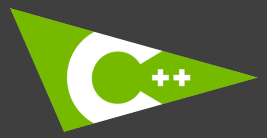
libcu++ (NVCC, NVC++)

Coming soon:

- nvc++ -stdlib=libstdc++ : std:: is host only (GCC ABI)
- nvc++ -stdlib=libcu++ : std:: is heterogeneous (NV ABI)

C++ standard library APIs and **CUDA-specific extensions**

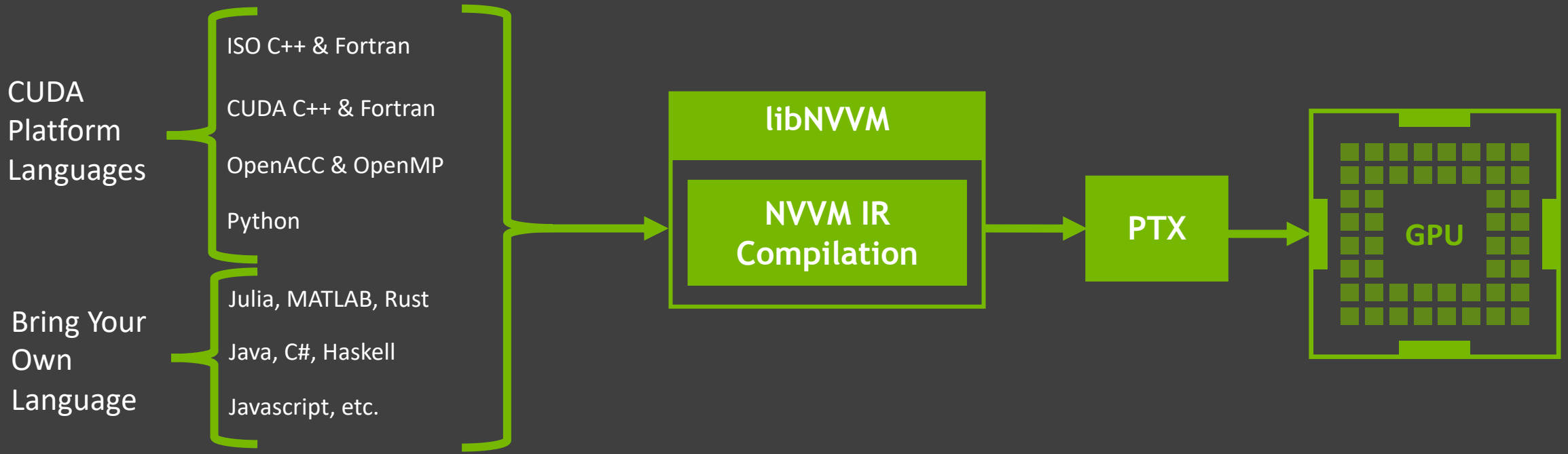
- Synchronization: atomic, atomic_ref, barrier, semaphores, …
- Time: chrono clocks, durations, rations, dates, calendars, …
- Utilities: type traits, array, tuple, complex, byte, …
- **Thread scopes:** thread, block, device, system
  - atomic<T, thread_scope>, barrier<thread_scope>, …
- CUDA extensions:
  - Asynchronous data movement: memcpy_async
  - Synchronization primitives: pipeline
  - Temporal locality & address spaces: annotated_ptr
  - Memory resources and allocators: memory_resource
- C++2x extensions



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

Open Source: https://github.com/NVIDIA/libcudacxx

NVC++: NVIDIA C++ COMPILER Built using libNVVM



NVC++: NVIDIA C++ COMPILER
Unsupported heterogeneous C++ features

- Annotations required for separate compilation.

```
// File a.cpp
__host__ __device__
int square(int) { … }

// File b.cpp
#include <a.hpp>
std::transform(par, begin(x), end(x), [&](int& x) {
    return square(x);
});
```

NVC++: NVIDIA C++ COMPILER
Unsupported heterogeneous C++ features

- Heterogeneous function pointers (WIP)
- Heterogeneous virtual functions
- Exceptions on device targets
- Syscalls on device targets

NVC++: NVIDIA C++ COMPILER
Unified memory

- Coherent-platforms (P9+V100, GH, CSCS Alps): all system-allocated memory is accessible from all host and device threads.

```
double GLOBAL;
void example(double a) {
    vector<double> x;
    auto ints = views::iota(0, x.size());
    std::for_each(par, ints.begin(), ints.end(),
       [&](int idx){ x[idx] += GLOBAL * a; }
   );
}
```

NVC++: NVIDIA C++ COMPILER
Unified memory

- Coherent-platforms (P9+V100, GH, CSCS Alps): all system-allocated memory is accessible from all host and device threads.
- Non-coherent platforms (PCI-e GPUs): only dynamically-allocated memory from files compiled by nvc++ is accessible to device.
- The following is also not accessible from GPU:
  - Globals
  - Host thread stacks
  - Files compiled with a different toolchain (e.g. external libraries)

```
double GLOBAL;
void example(double a) {
    vector<double> x;
    auto ints = views::iota(0, x.size());
    std::for_each(par, ints.begin(), ints.end(),
       [&](int idx){ x[idx] += GLOBAL * a; }
   );
}
```

warning: function "**lambda [](int)->void**" captures local object "**a**" by reference, will likely cause an illegal memory access when run on the device.

NVC++: NVIDIA C++ COMPILER
Unified memory

- Coherent-platforms (P9+V100, GH, CSCS Alps): all system-allocated memory is accessible from all host and device threads.
- Non-coherent platforms (PCI-e GPUs): only dynamically-allocated memory from files compiled by nvc++ is accessible to device.
- The following is also not accessible from GPU:
  - Globals
  - Host thread stacks
  - Files compiled with a different toolchain (e.g. external libraries)

```
double GLOBAL;
void example(double a) {
    vector<double> x;
    auto ints = views::iota(0, x.size());
    std::for_each(par, ints.begin(), ints.end(),
       [x = x.data(), GLOBAL, a](int idx) {
            x[idx] += GLOBAL * a;
    });
}
```

Capture by value (=) to copy data to the device.

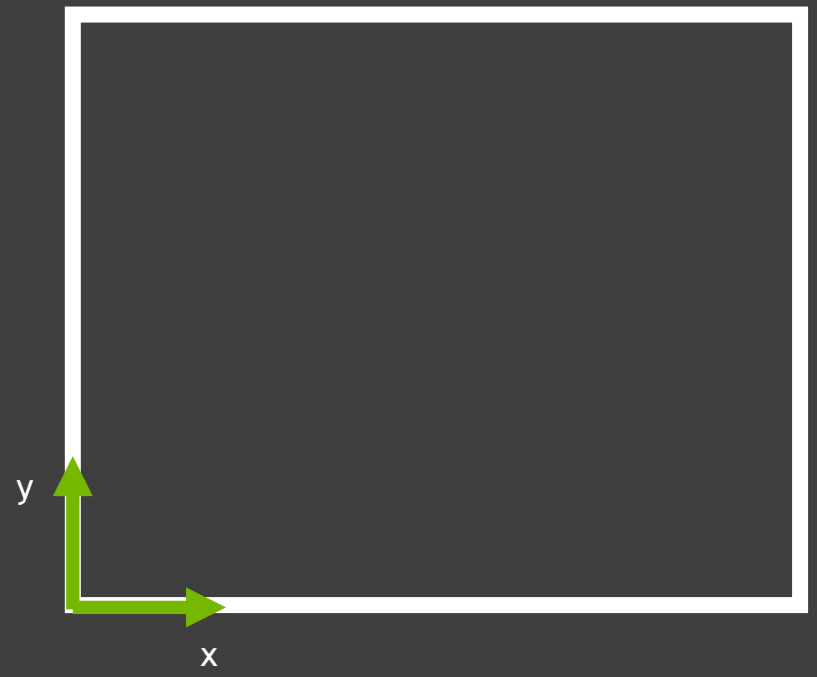
NVC++: NVIDIA C++ COMPILER References

- NVC++ Parallel Algorithms Guidelines: https://docs.nvidia.com/hpc-sdk/compilers/c++-parallel-algorithms/index.html#guidelines

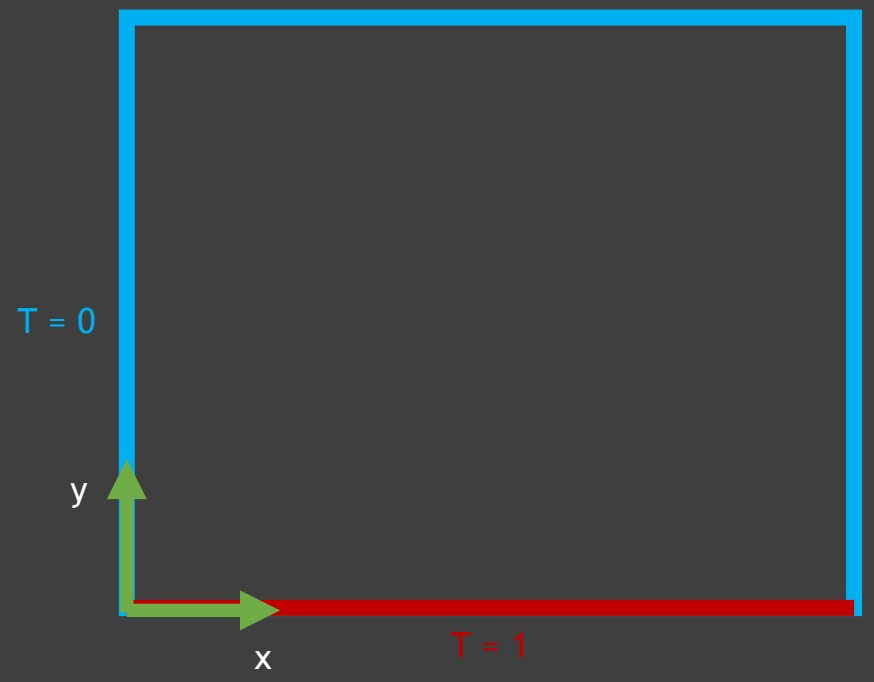
- Bryce Lelbach, *Inside NVC++ and NVFORTRAN*, GTC'21 Spring

Lab 2: 2D unsteady heat-equation (Part I)

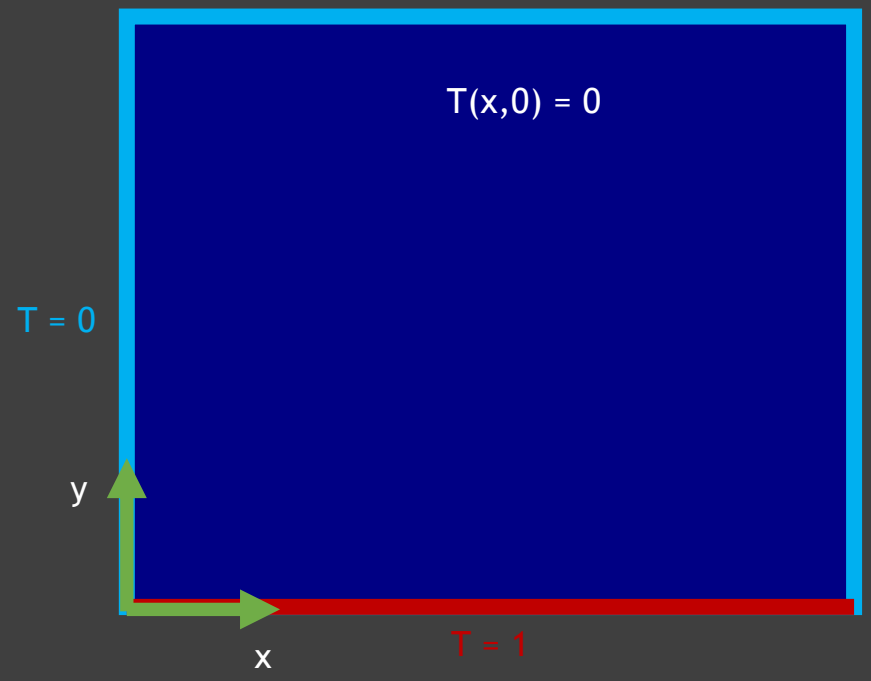
2D unsteady heat equation
Problem description



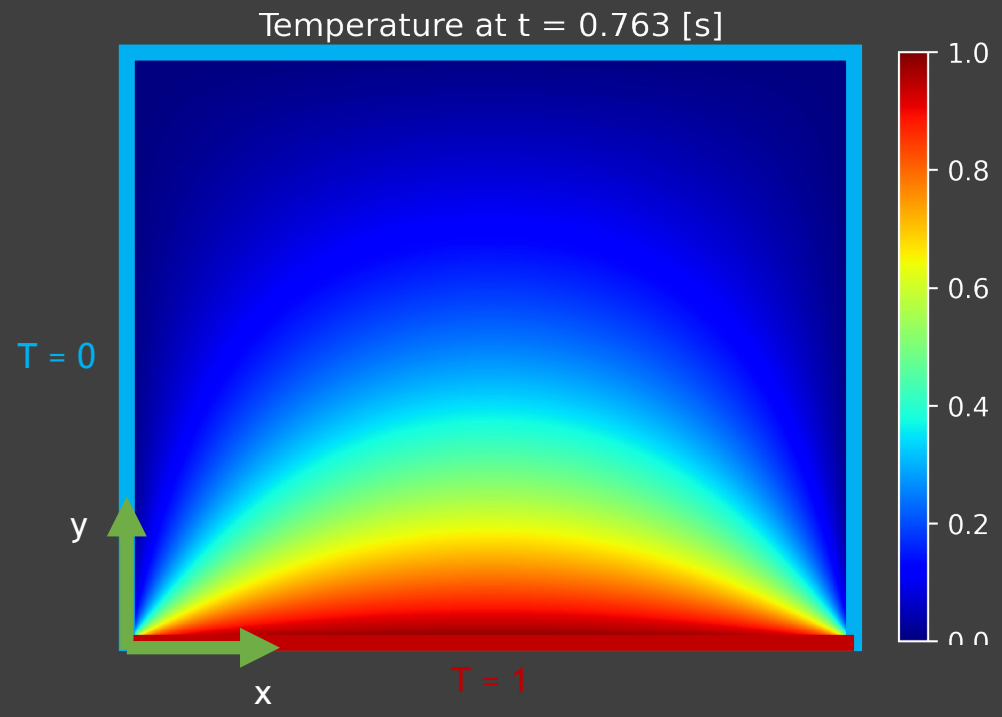
2D unsteady heat equation
Problem description



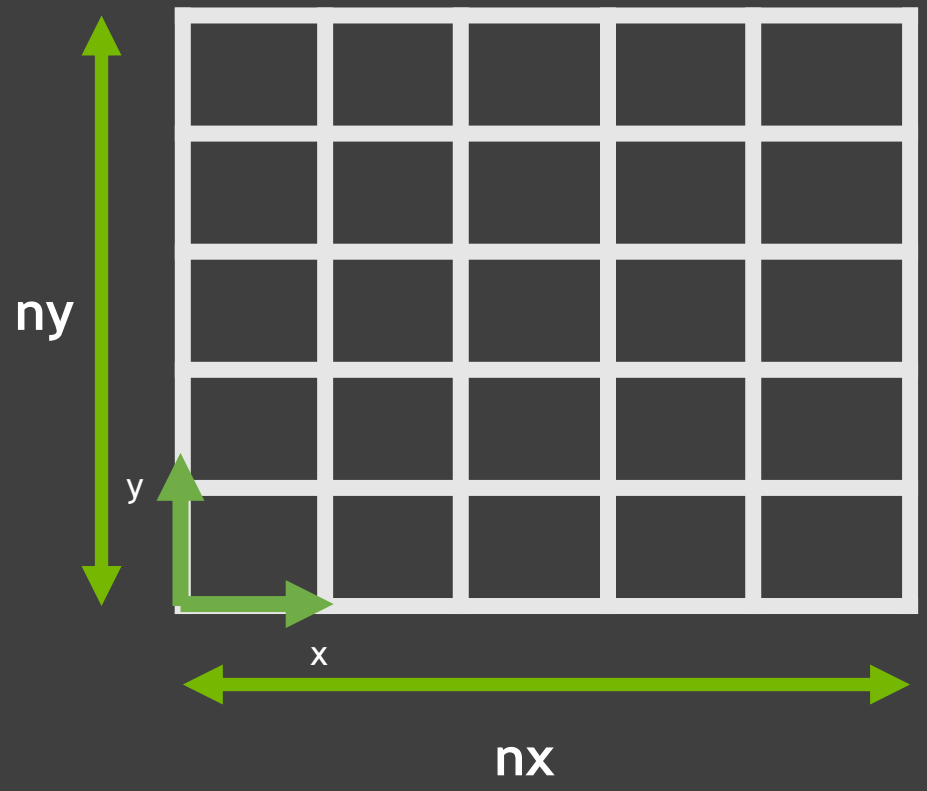
2D unsteady heat equation
Problem description



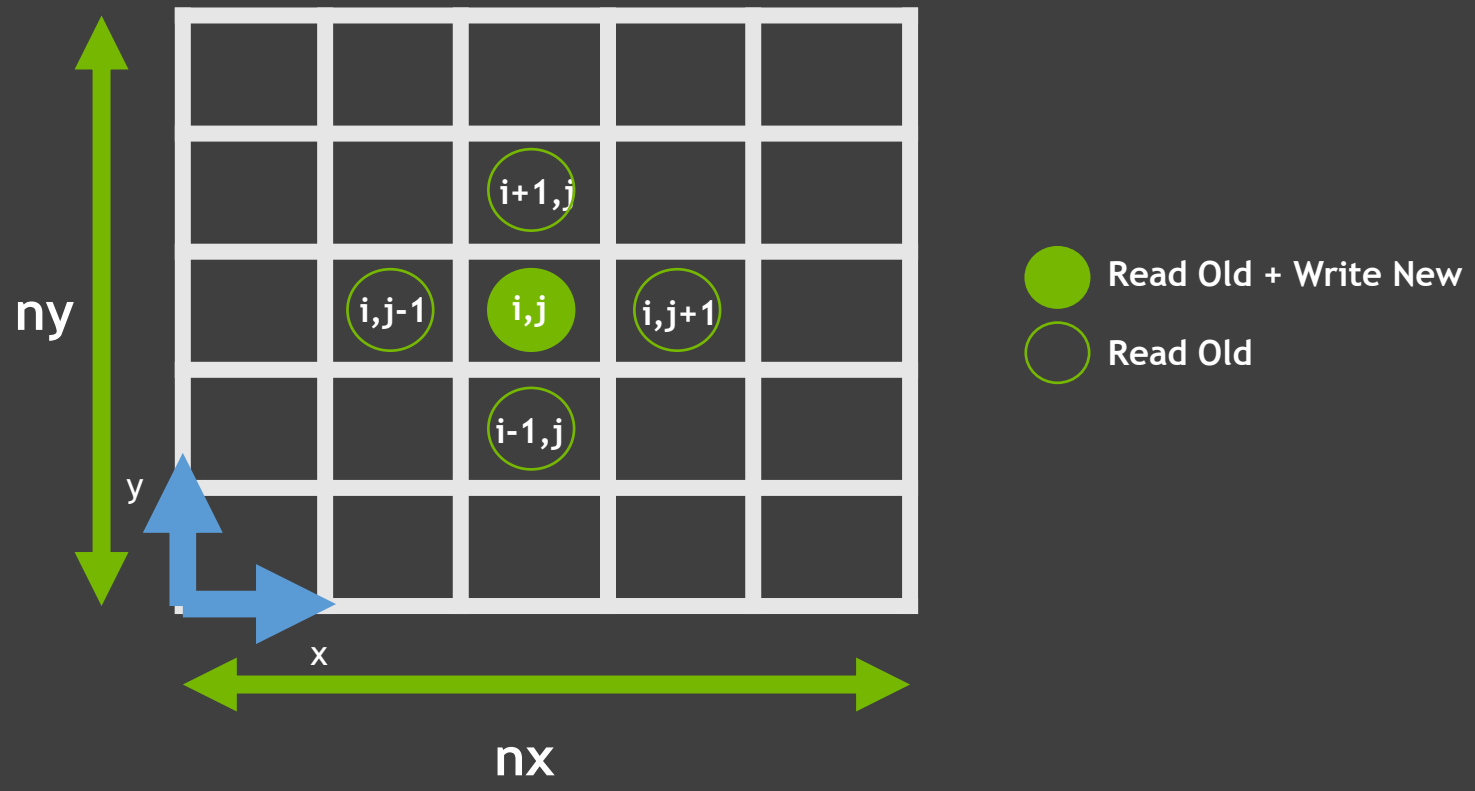
2D unsteady heat equation
Problem description



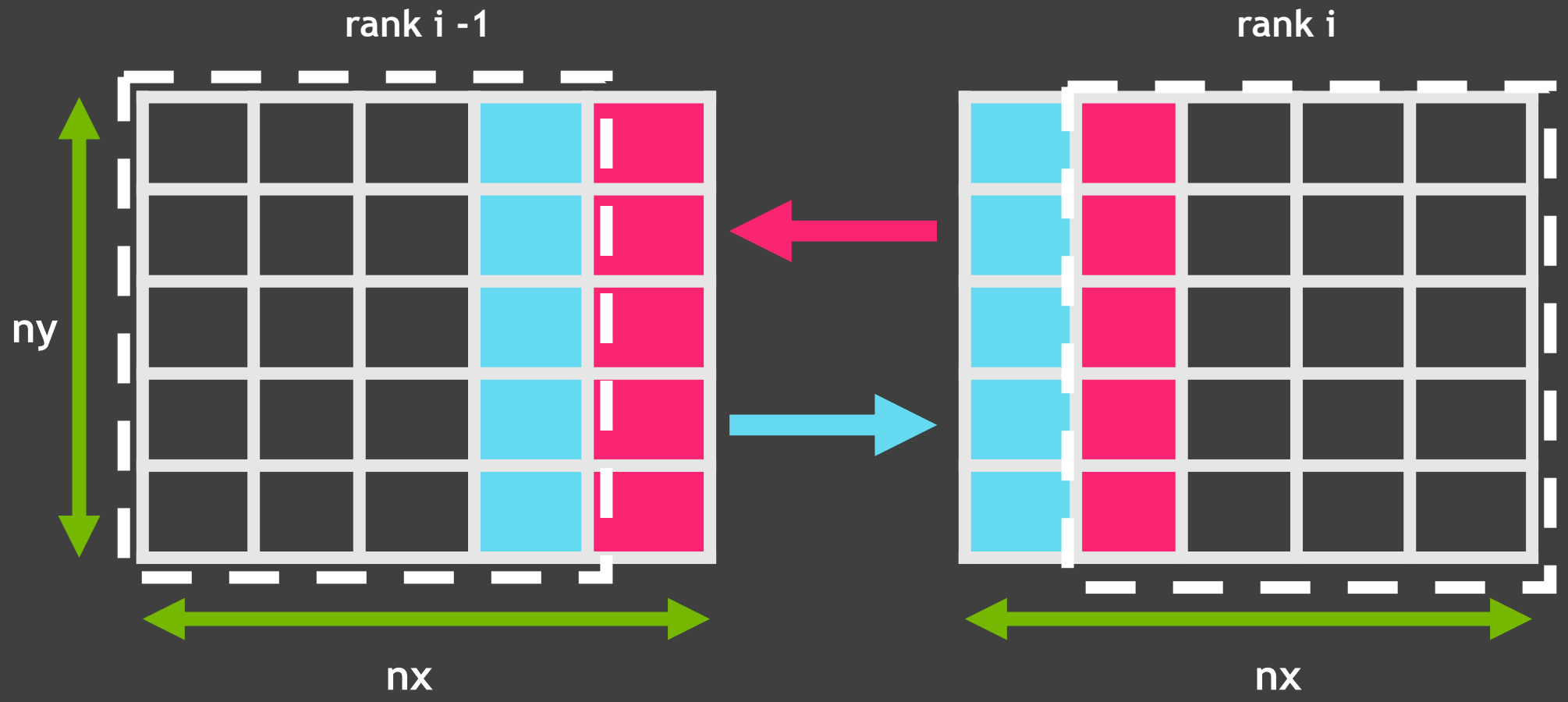
2D unsteady heat equation
Discretization



2D unsteady heat equation Stencil



2D unsteady heat equation
Domain decomposition



2D unsteady heat equation
Stencil: updates one element at (x,y)

```
double stencil(double* u_new, double* u_old, long x, long y, params p) {
  apply_boundary_conditions(u_old, x, y, p);

  u_new[fuse(x, y)] = stencil(u_old, x, y, p);

  return u_new[fuse(x, y)] * p.dx * p.dx;
}
```

2D unsteady heat equation
Grid kernel: updates all elements

```
struct grid { long x_start, x_end, y_start, y_end; };

double stencil(double* u_new, double* u_old, grid g, params p) {
  double energy = 0.;
  for (long x = g.x_start; x < g.x_end; ++x) {
    for (long y = g.y_start; y < g.y_end; ++y) {
      energy += update_one(u_new, u_old, x, y, p);
    }
  }
  return energy;
}
```

2D iteration
Raw nested loops

This raw nested loop....

```
double energy = 0.;
for (long x = g.x_start; x < g.x_end; ++x) {
  for (long y = g.y_start; y < g.y_end; ++y) {
    energy += stencil(u_new, u_old, x, y, p);
  }
}
```

...can be written using sequential algorithms...

```
double energy = transform_reduce(g.x_start, g.x_end, plus<>,
    [&](long x) {
       return transform_reduce(g.y_start, g.y_end, plus<>,
         [&](long x) { return stencil(u_new, u_old, x, y, p); }
  });
```

2D iteration
Nested parallel algorithms

Parallel algorithms can be nested...

```
double energy = transform_reduce(par, g.x_start, g.x_end, plus<>,
    [&](long x) {
       return transform_reduce(par, g.y_start, g.y_end, plus<>,
         [&](long x) { return stencil(u_new, u_old, x, y, p);  }
  });
```

...but some implementations do not support "nested parallelism": the ability to create work in parallel...

…so the inner algorithm calls run sequentially on those.

2D iteration
Flat parallel algorithms

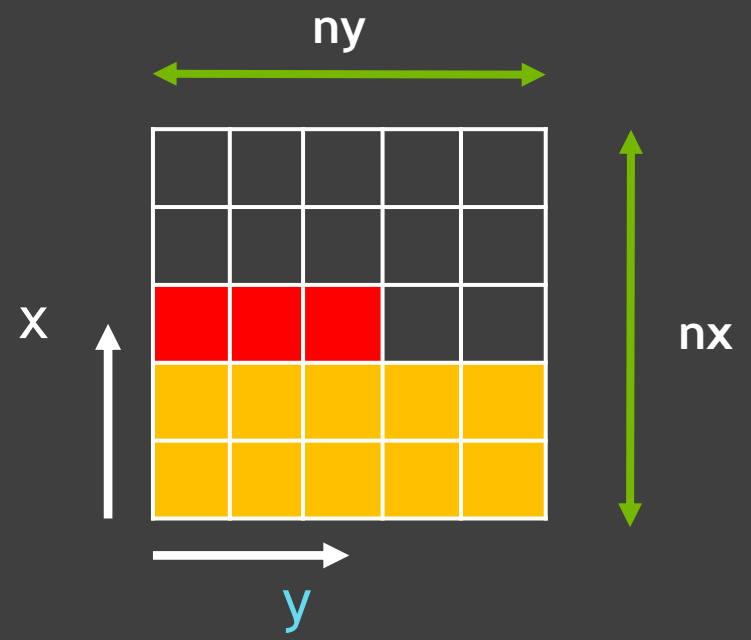
When possible, it is often more efficient to avoid nested parallelism...

```
double energy = transform_reduce(par, ???, ???, plus<>, [&](long x) {
    return stencil(u_new, u_old, x, y, p);
});
```

...by mapping multi-dimensional iteration into one-dimensional iteration...

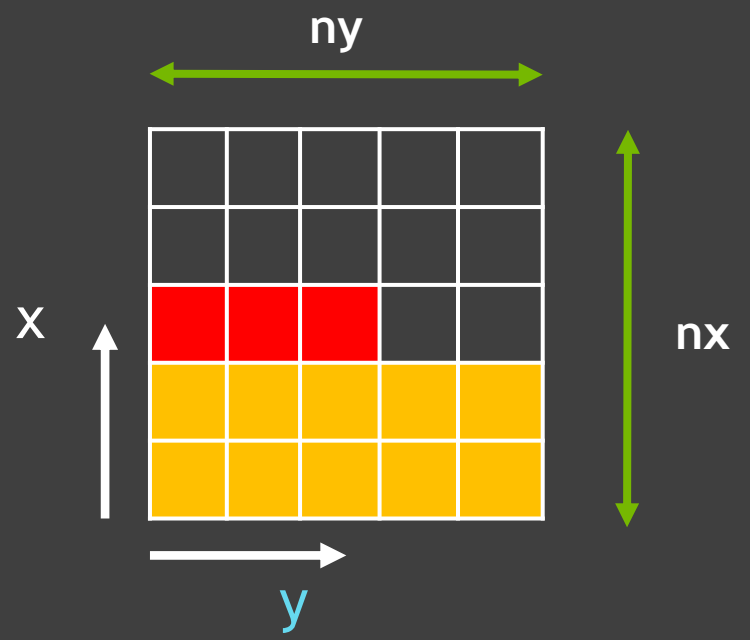
C++17: one-dimensional iteration fuse: from 2D to 1D

```
// Indexing invariant
auto [x1, y1] = split(fuse(x0, x1));
assert(x0 == x1 && y0 == y1);

// 2D indices -> 1D index
long fuse(long x, long y) {
    return x * ny + y;
};
```



C++17: one-dimensional iteration split: from 1D to 2D

```
// Indexing invariant
auto [x1, y1] = split(fuse(x0, x1));
assert(x0 == x1 && y0 == y1);

// 1D index -> 2D indices
std::pair<long, long> split(long i) {
    return {
        i / ny + x_start,
        i % ny + y_start
    };
};
```



C++17: one-dimensional iteration Kernel launch

```
// Two-dimensional domain
long N = nx * ny;

// One dimensional iteration
std::for_each_n(ints.begin(), N, [u_new, u_old,…](auto i) { // 1D idx
        // Recover 2D index:
        auto [x, y] = split(i);
        stencil(u_new, u_old, x, y);
});
```

C++23: two dimensional iteration views::cartesian_product

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

std::for_each(stde::par, v.begin(), v.end(), [](auto& e) {
    auto [i, j] = e;
});
```

C++23: two-dimensional iteration Kernel launch

```
// Two-dimensional domain
auto xs = stdv::iota(g.x_start, g.x_end); // Lazy ranges of integers
auto ys = stdv::iota(g.y_start, g.y_end);
auto r = stdv::cartesian_product(xs, ys); // [(0,0), (0,1), …(n,n)]

// Two dimensional iteration
std::for_each(r.begin(), r.end(), [u_new, u_old,…](auto i) { // 2D idx
        // Recover 2D indices:
        auto [x, y] = i;
        stencil(u_new, u_old, x, y);
});
```

Multi-dimensional iteration Summary

- C++17 and C++20: one-dimensional iteration (see previous section)
  - Create a one-dimensional range using pointers, counting iterators, views::iota, etc.
  - Create fuse/split functions to map 1D -> 2D indices and vice-versa
  - Use the split function within the lambda to retrieve the 2D index from the 1D iteration

- C++23: multi-dimensional iteration with views::cartesian_product
  - Create one one-dimensional range per dimension

```
auto xs = stdv::iota(nx);
auto ys = stdv::iota(ny);
```

  - Create a multi-dimensional range with views::cartesian_product

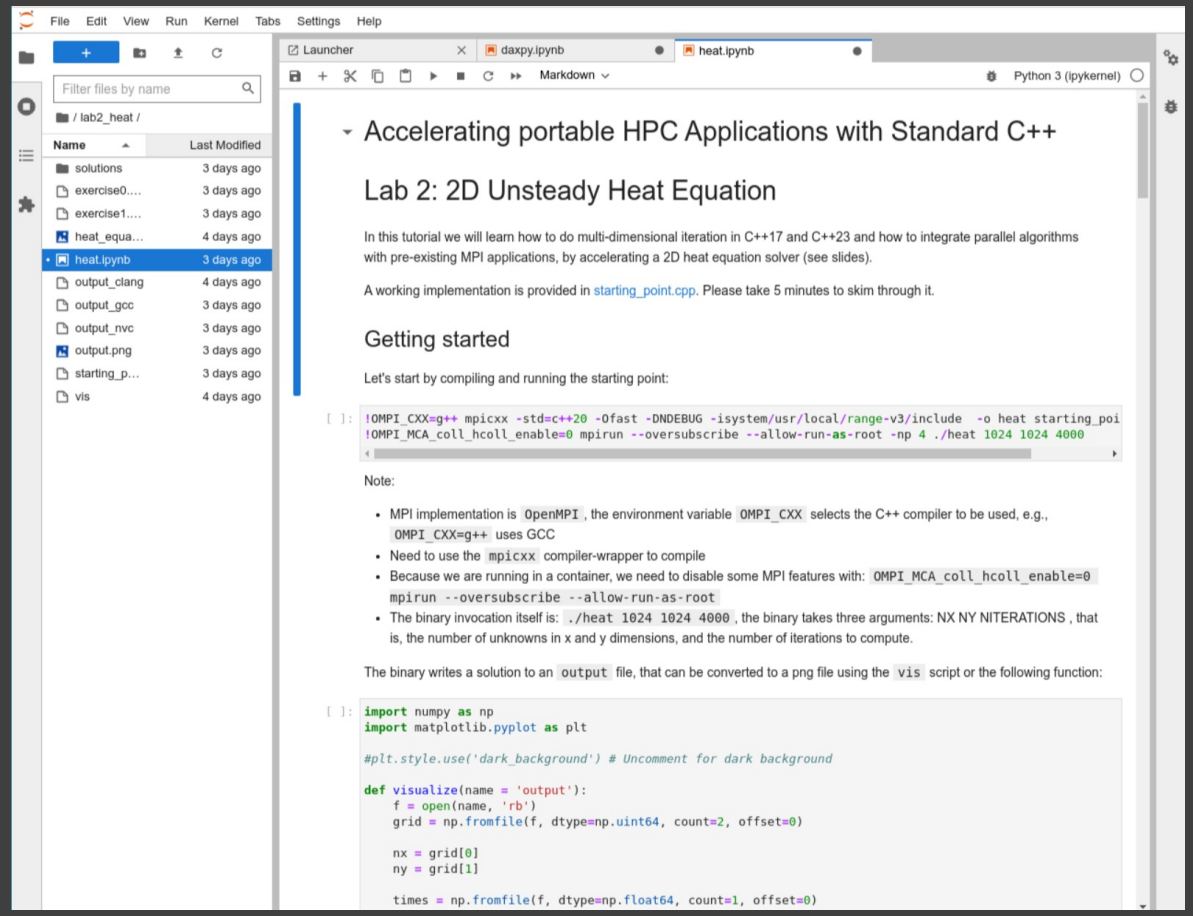
```
auto r = stdv::cartesian_product(xs, ys);
```

  - Within the lambda, easily extract the multi-dimensional indices

```
auto [x, y] = i;
```

Lab 2: 2D heat equation (Part I)

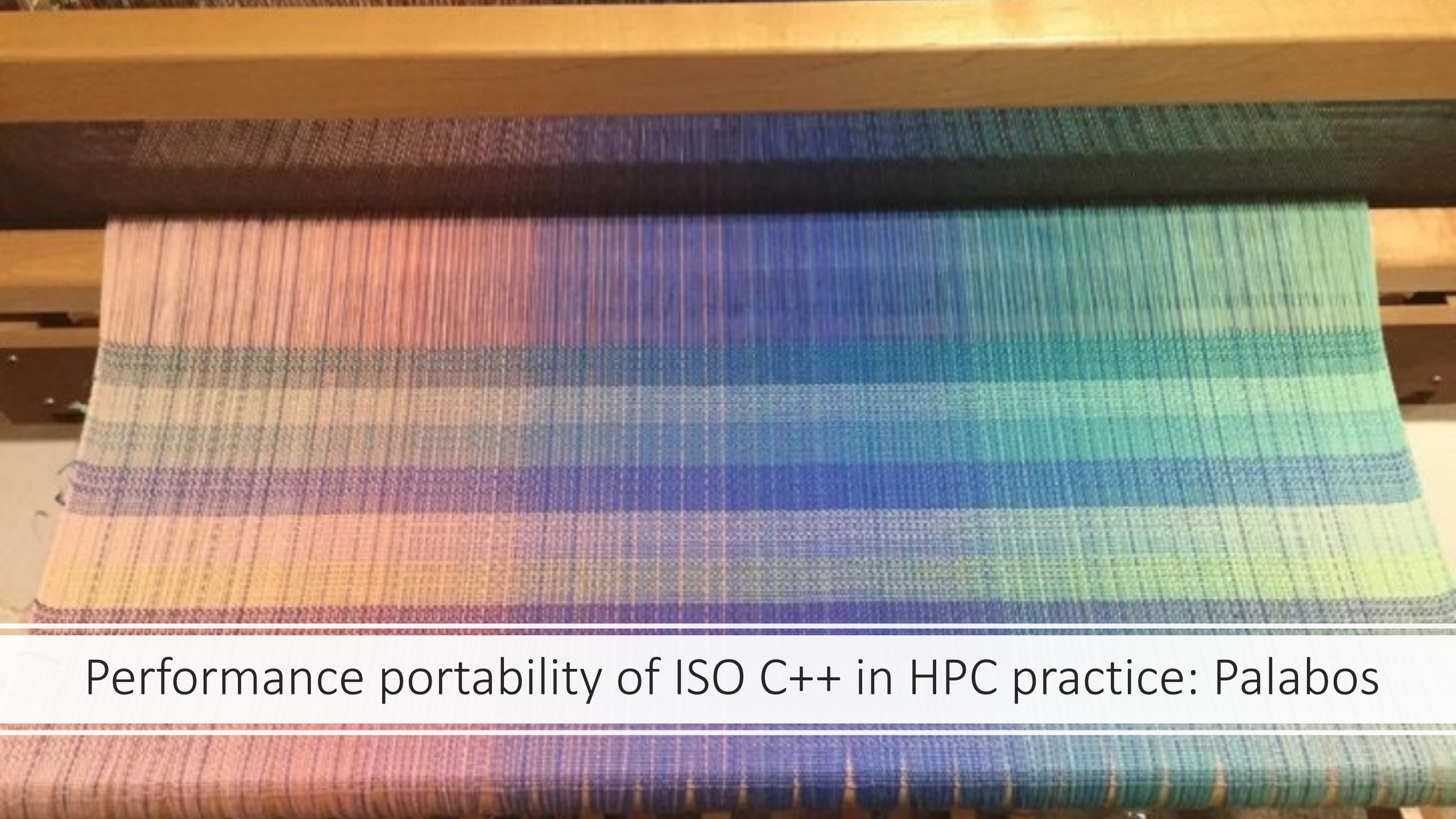
- **Exercise 1:** parallelize the MPI implementation using the STL parallel algorithms with the different types of indexing to create an hybrid MPI/C++ application.



Lab 2: 2D heat equation Solutions (DEMO)

2D unsteady heat equation Solution

```
double stencil(double* u_new, double* u_old, grid g, params p) {
  auto xs = stdv::iota(g.x_start, g.x_end); // Lazy ranges of integers
  auto ys = stdv::iota(g.y_start, g.y_end);
  auto r = stdv::cartesian_product(xs, ys); // [(0,0), (0,1), …(n,n)]

  return std::transform_reduce(
    stde::par, r.begin(), r.end(), 0., std::plus{}, [=](auto idx) {
       auto [x, y] = idx;
       return stencil(u_new, u_old, x, y, p);
  });
}
```



Performance portability of ISO C++ in HPC practice: Palabos

Palabos: Multi-physics simulation framework

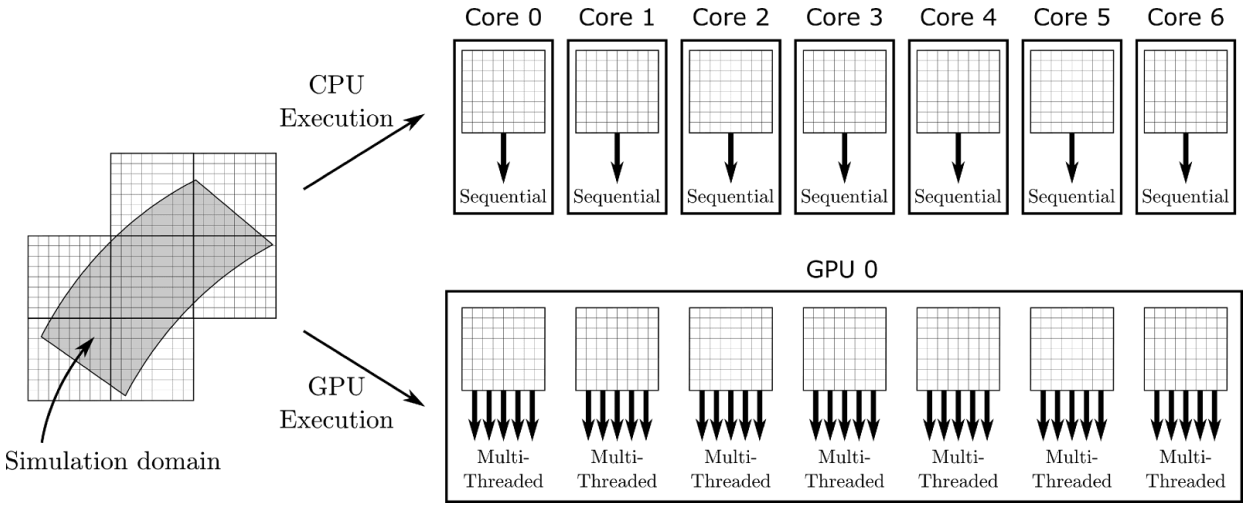
- Developed at University of Geneva
- Based on Lattice Boltzmann methods
- Multi-phase flows, flow through porous media, etc.

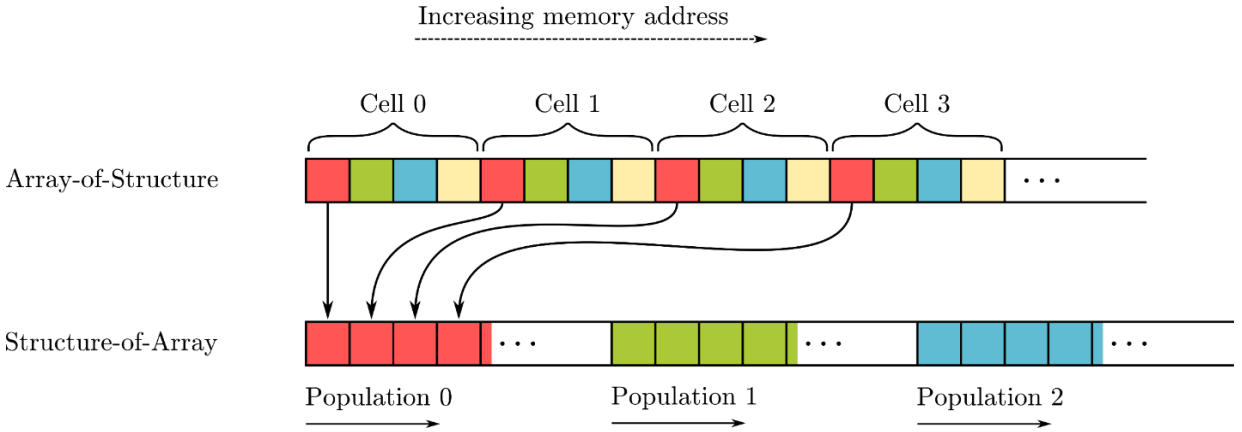
- C++ Codebase with ~100k LOC
- **Programming model:** MPI/C++
- **Reuse MPI backend to get a Multi-GPU version**

Palabos Algorithms

```
std::for_each(par_unseq, begin(c), end(c),
[c = c.data()](double& i) {
  int i = &x - v_ptr;

  double f_local[19];
  for (int k = 0; k < 19; ++k)
    f_local[k] = f[i][k];

  collide(f_local);

  auto [iX, iY] = split(i);
  for (int k = 0; k < 19; ++k) {
     int nb = fuse(iX+c[k][0], iY+c[k][1]);
      ftmp[nb][k] = f_local[k];
  }
});
```

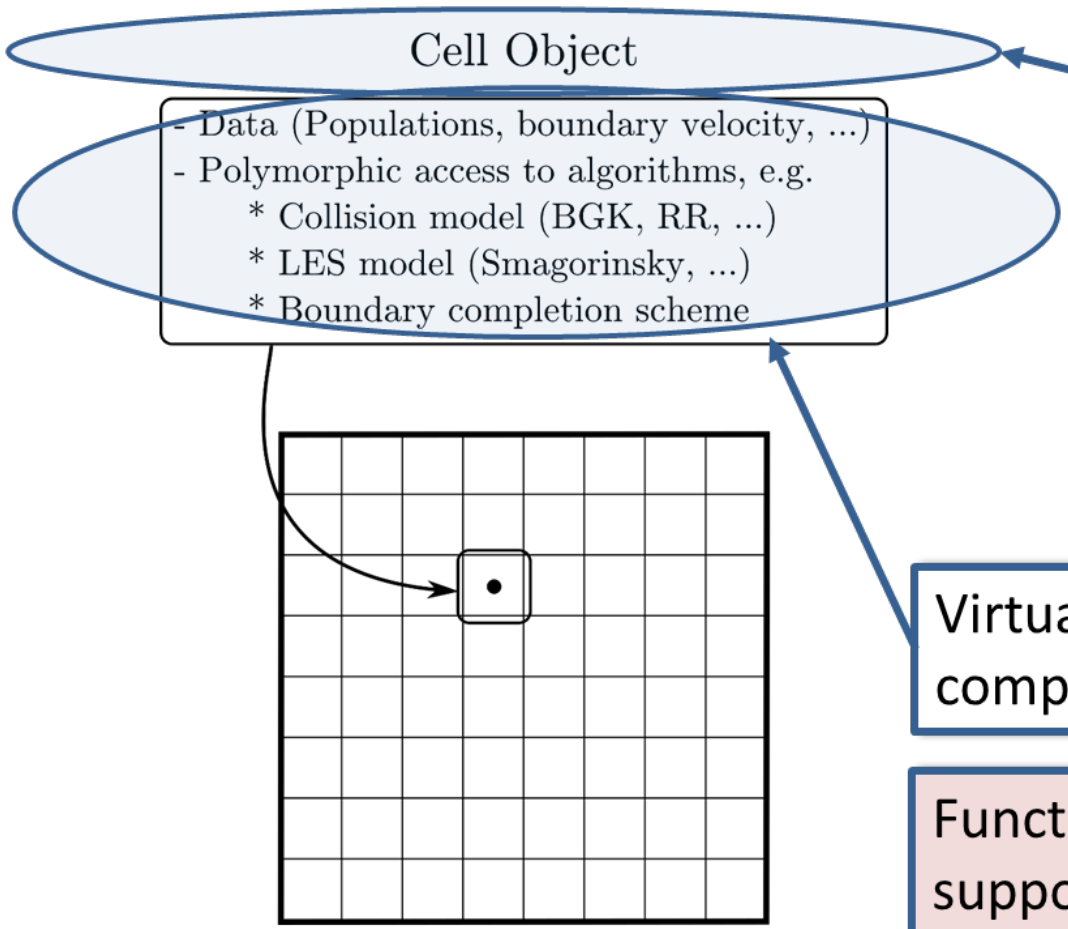
- std::for_each for processing grid cells
- std::transform_reduce: for reductions while processing cells
- std::exclusive_scan for I/O: file offsets, packing/unpacking buffer offset, etc.





Palabos: Object-oriented approach

Polymorphism allows every grid node to implement different physical / numerical model



Cell objects contain local data, typically 19 floats, the “populations”.

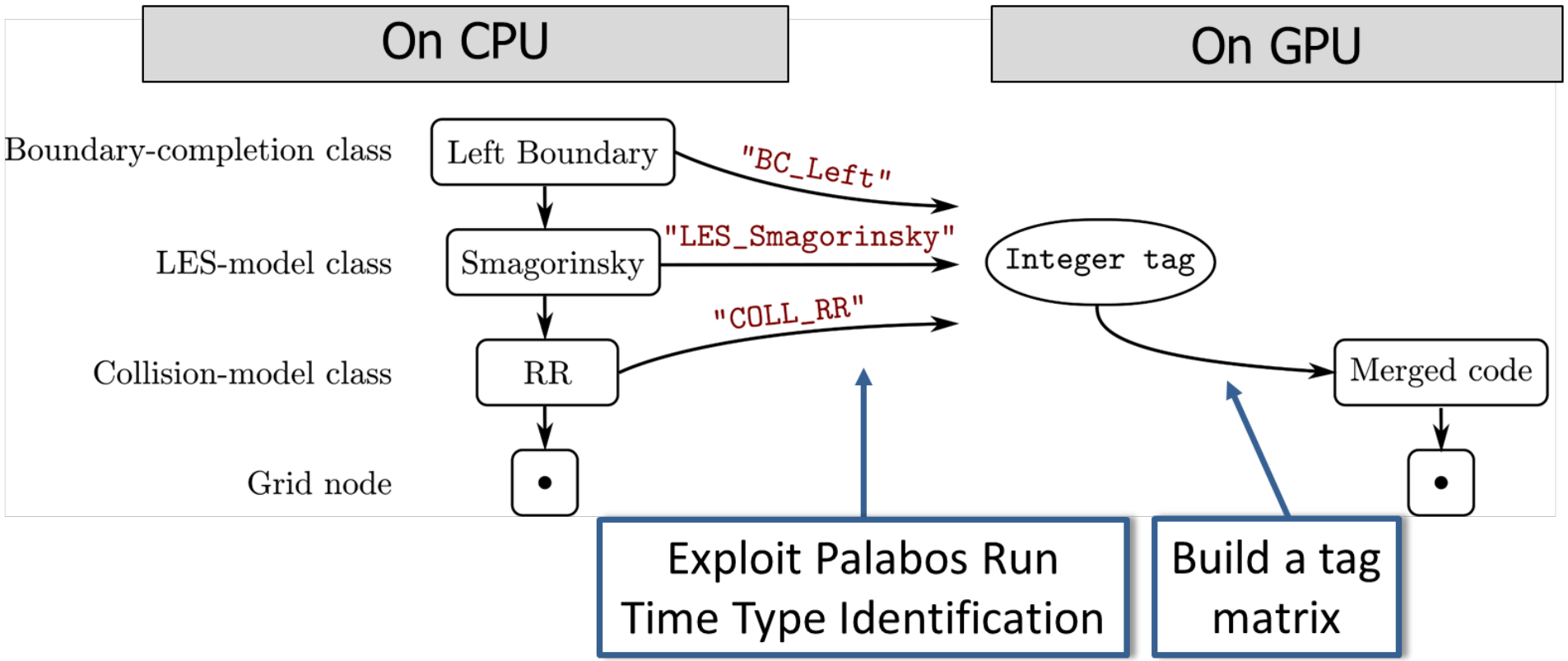
GPUs don’t like the resulting memory layout.

Virtual function calls to different model components.

Function-pointer call mechanism is not supported.

Polymorphic objects → Tag matrix

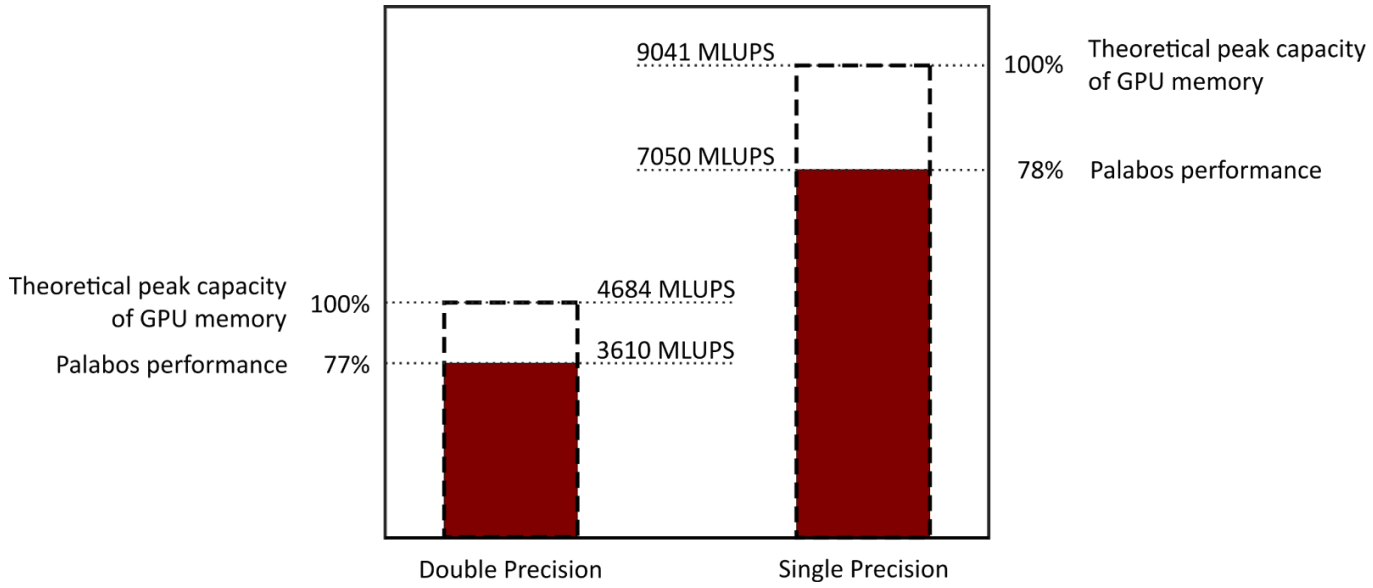
The tag matrix suits the GPU better and can be generated automatically



Palabos
1GPU Performance

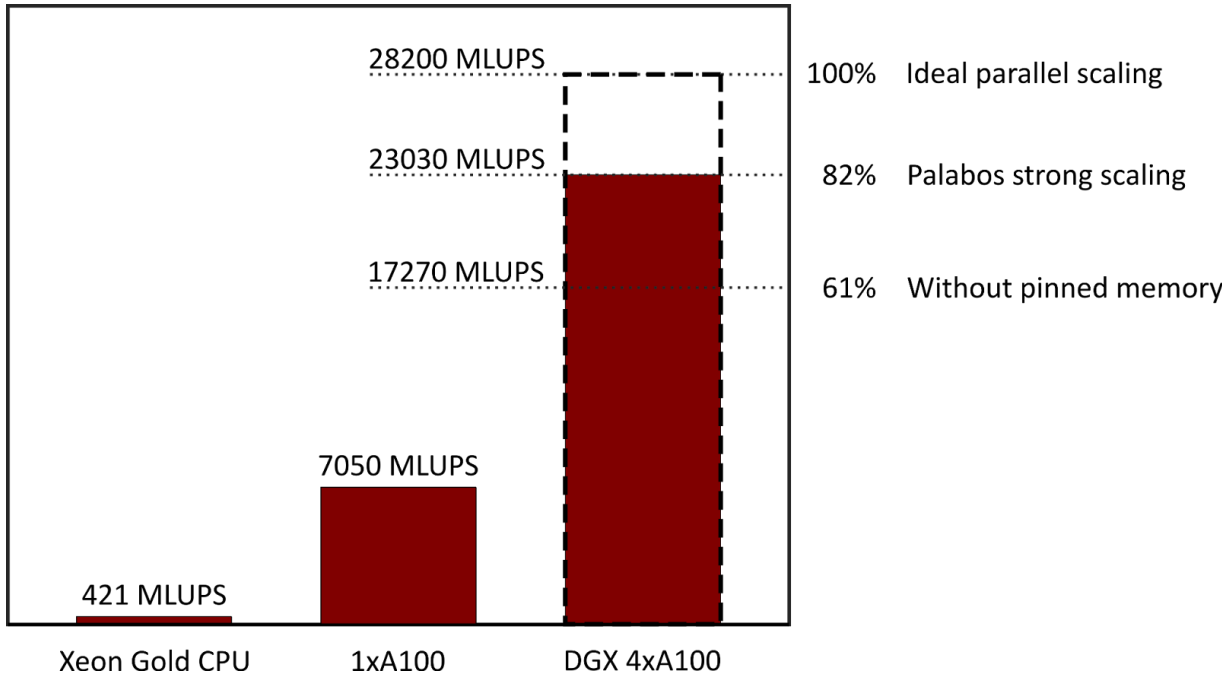
Performance model:

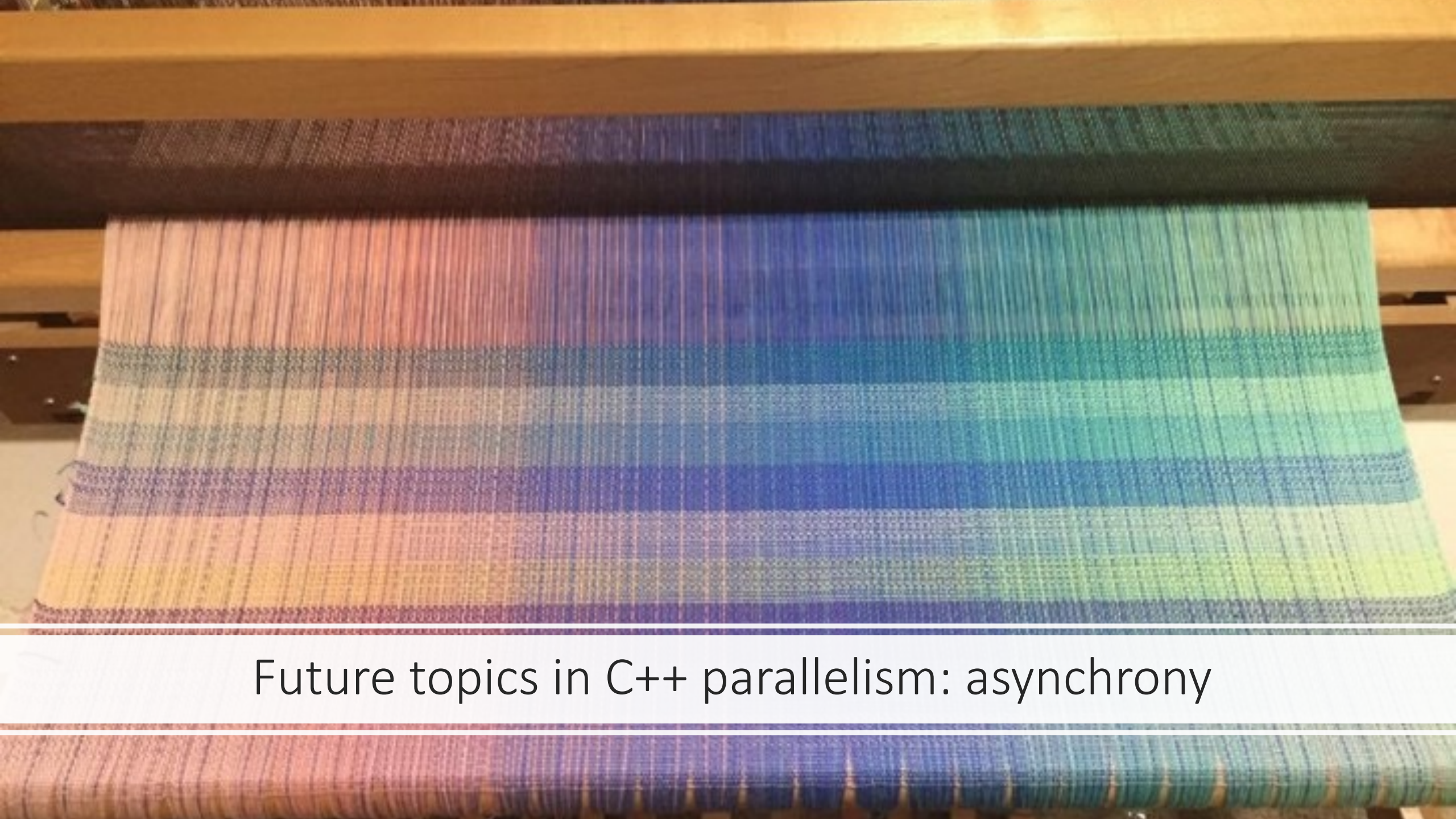
- Lattice Boltzmann D2Q19:
  19 fields + 1 index
- Memory bound
- Mega Lattice Updates / s:
  - BW_DRAM / (2 * 20 * Float Bytes)
  - A100-40: ~1500 GB/s / (2 * 20 * 8 B) ~= 4684 MLUPs
- State of the art: ~80% SOL



Palabos
4 GPU Performance

- DGX Station A100
- Difference between using pinned vs unified memory for packing/unpacking buffers
- On going: weak scaling on Selene.
- On going: SoftwareX publication, and blogpost.
- Near future: completely overlap communication with computation.

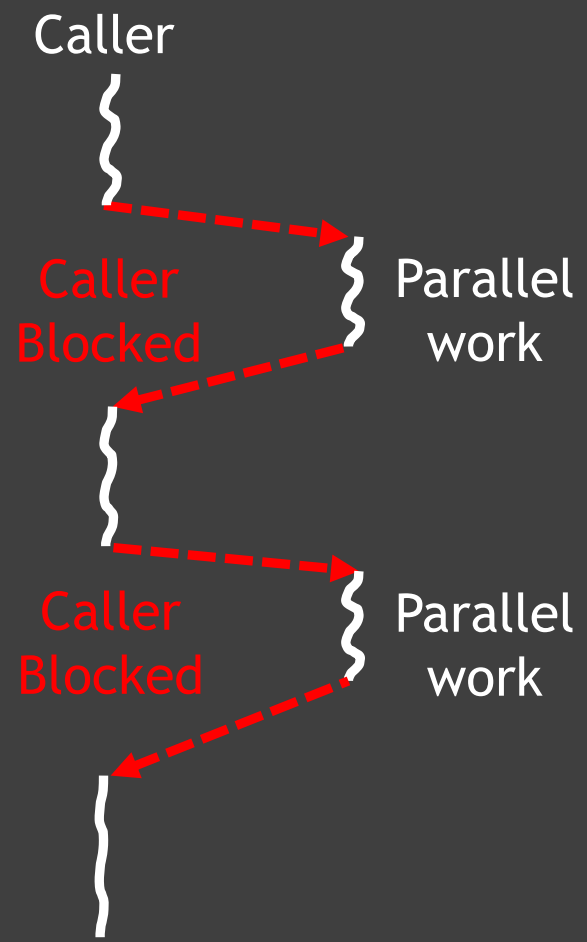




Future topics in C++ parallelism: asynchrony

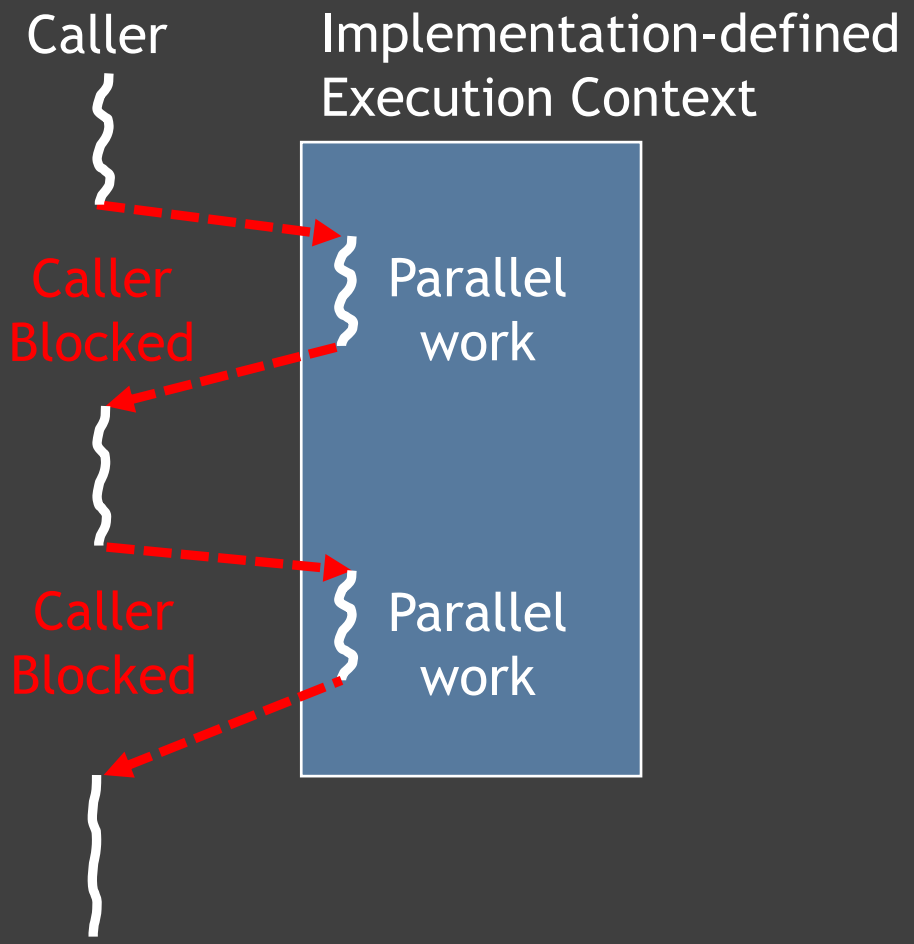
Limitations of parallel algorithms
Fork-join synchronous

```
std::vector<int> x {…};

std::sort(stde::par, x.begin(), x.end());


std::unique(stde::par, x);
```

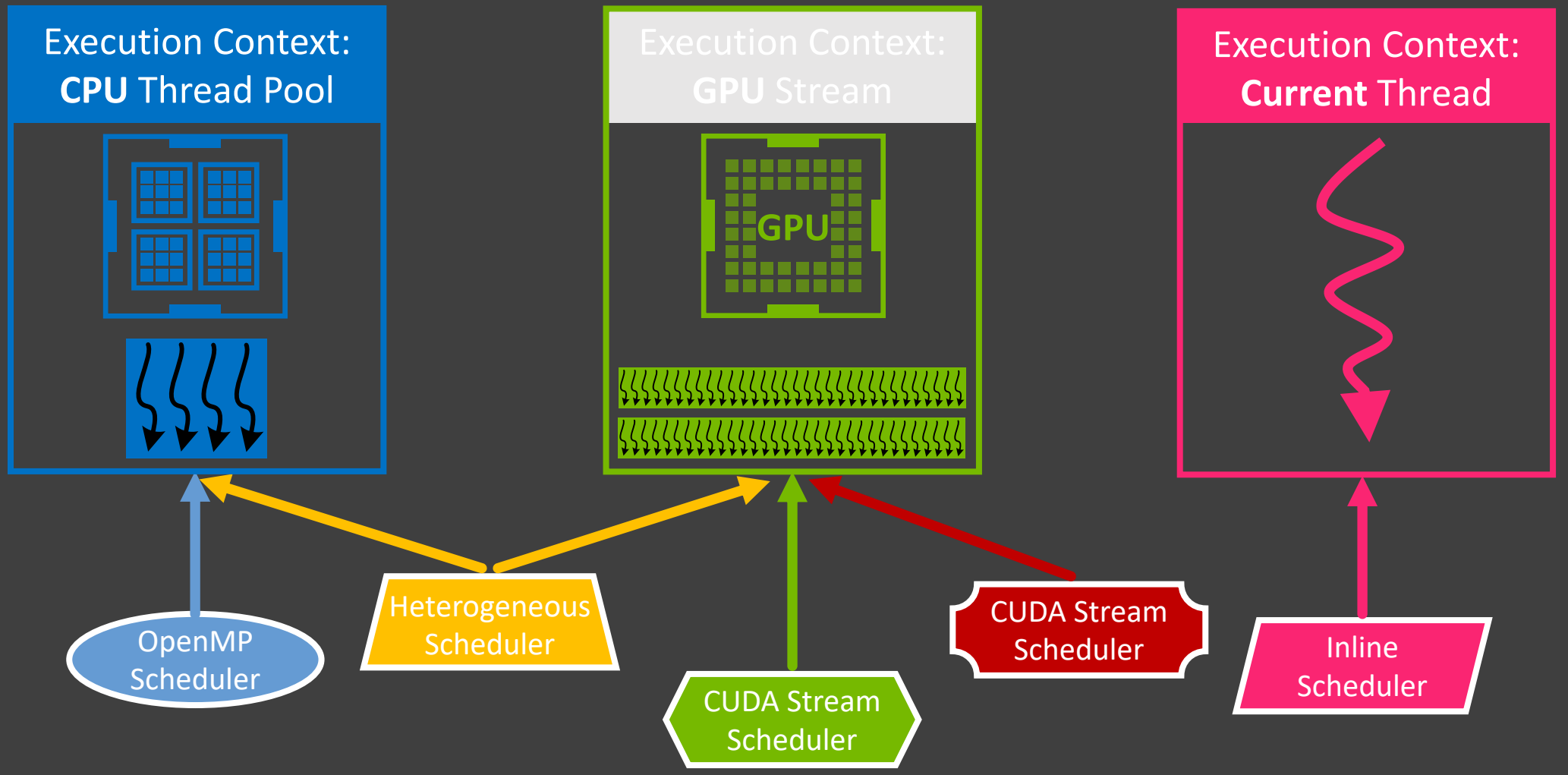


Limitations of parallel algorithms
Fork-join synchronous

```
std::vector<int> x {…};

std::sort(stde::par, x.begin(), x.end());

std::unique(stde::par, x);
```

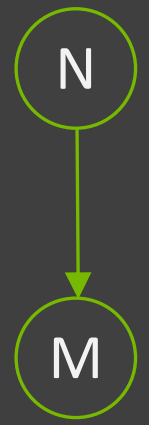


P2300R4: std::execution



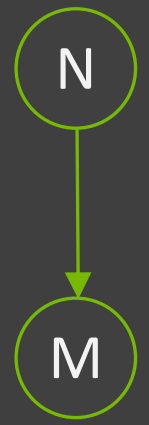
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



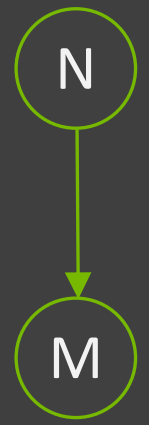
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



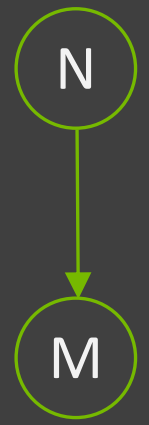
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



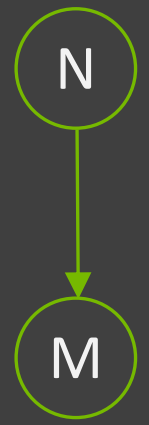
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



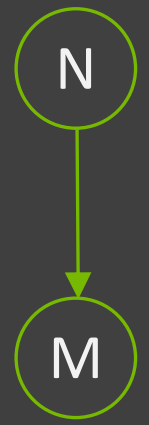
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



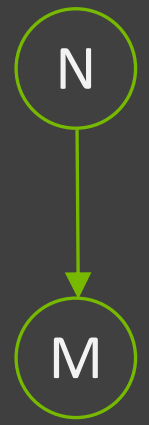
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



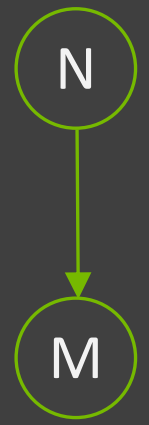
P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();          // Obtain scheduler from somewhere

sender auto begin = std::execution::schedule(sch);     // Handle to send work to scheduler
sender auto N = std::execution::then(begin, []() {     // Attach some work
    return 13;
});

sender auto M = std::execution::then(N, [](int arg) { // Attach more work
    return arg + 42;
});

                                                       // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();     // Start work; current thread waits
assert(r == 55);
```



P2300R4: std::execution

```
scheduler auto sch = thread_pool.scheduler();       // Obtain scheduler from somewhere

sender auto M = std::schedule(sch)                  // Handle to send work to scheduler
    | std::then([] { return 13; })                  // Attach some work
    | std::then([](int arg) {                       // Attach more work
          return arg + 42;
      });


                                                    // No work has started yet!
auto [r] = std::this_thread::sync_wait(M).value();  // Start work; current thread waits
assert(r == 55);
```

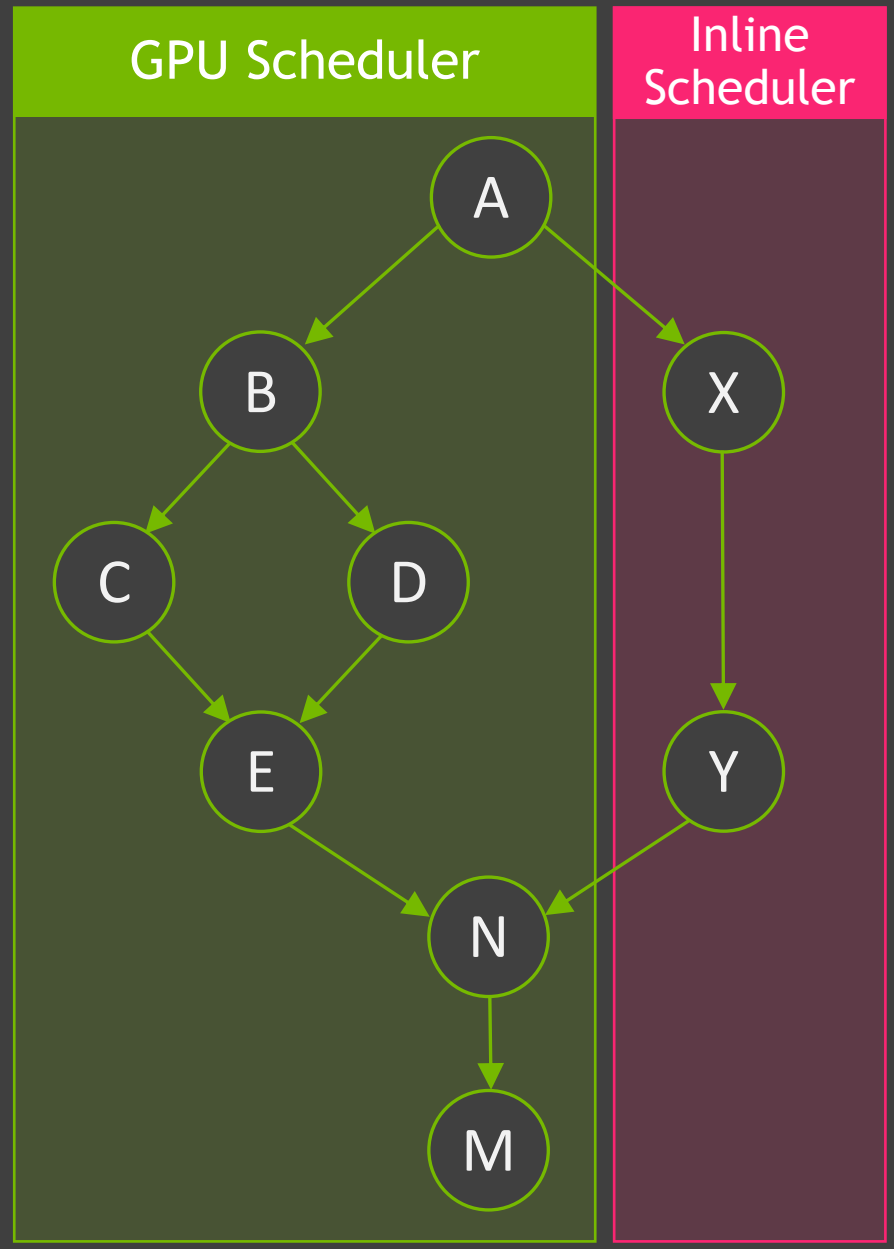


std::execution: P2300R4: std::execution Structured concurrency and parallelism

```
sender auto async_algo(sender auto s) {                    // sender adaptor
  return s | std::then([] { return 13; })
           | std::then([](int arg) { return arg + 42; });
}

scheduler auto sch = thread_pool.scheduler();              // Obtain scheduler from somewhere
sender auto M = async_algo(std::schedule(sch));            // Compose async algorithms
                                                           // No work has started yet!
auto [r] = std::this_thread::sync_wait(B).value();         // Start work; current thread waits
assert(r == 55);
```

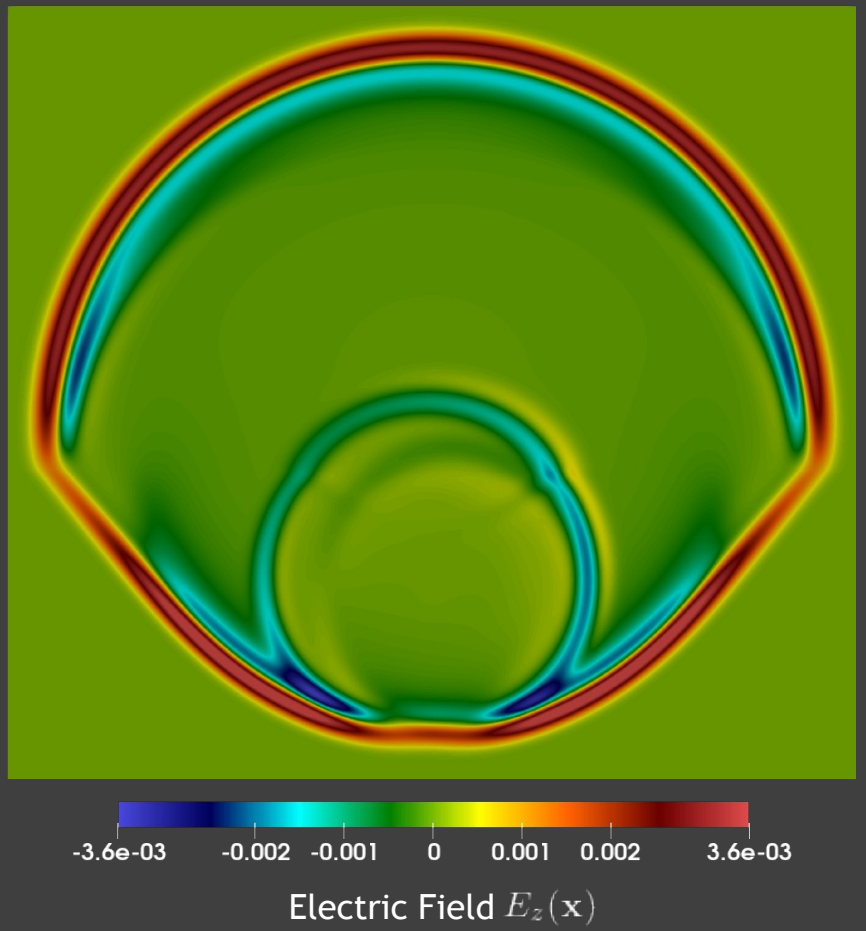
P2300R4: std::execution

```
sender auto async_graph(sender auto s) {
    auto A = s | std::then(printer{'A'}) | std::split();
    auto B = A | std::then(printer{'B'}) | std::split();
    auto C = B | std::then(printer{'C'});
    auto D = B | std::then(printer{'D'});
    auto E = std::when_all(C, D) | std::then(printer{'E'});
    auto X = A | std::then(printer{'X'})
               | std::then(printer{'Y'});
    return std::when_all(E, X);
}

cuda::execution::scheduler gpu_scheduler;
sender auto work = async_graph(async_algo(std::schedule(gpu_scheduler)));
auto [r] = std::this_thread::sync_wait(work).value();
assert(r == 55);
```



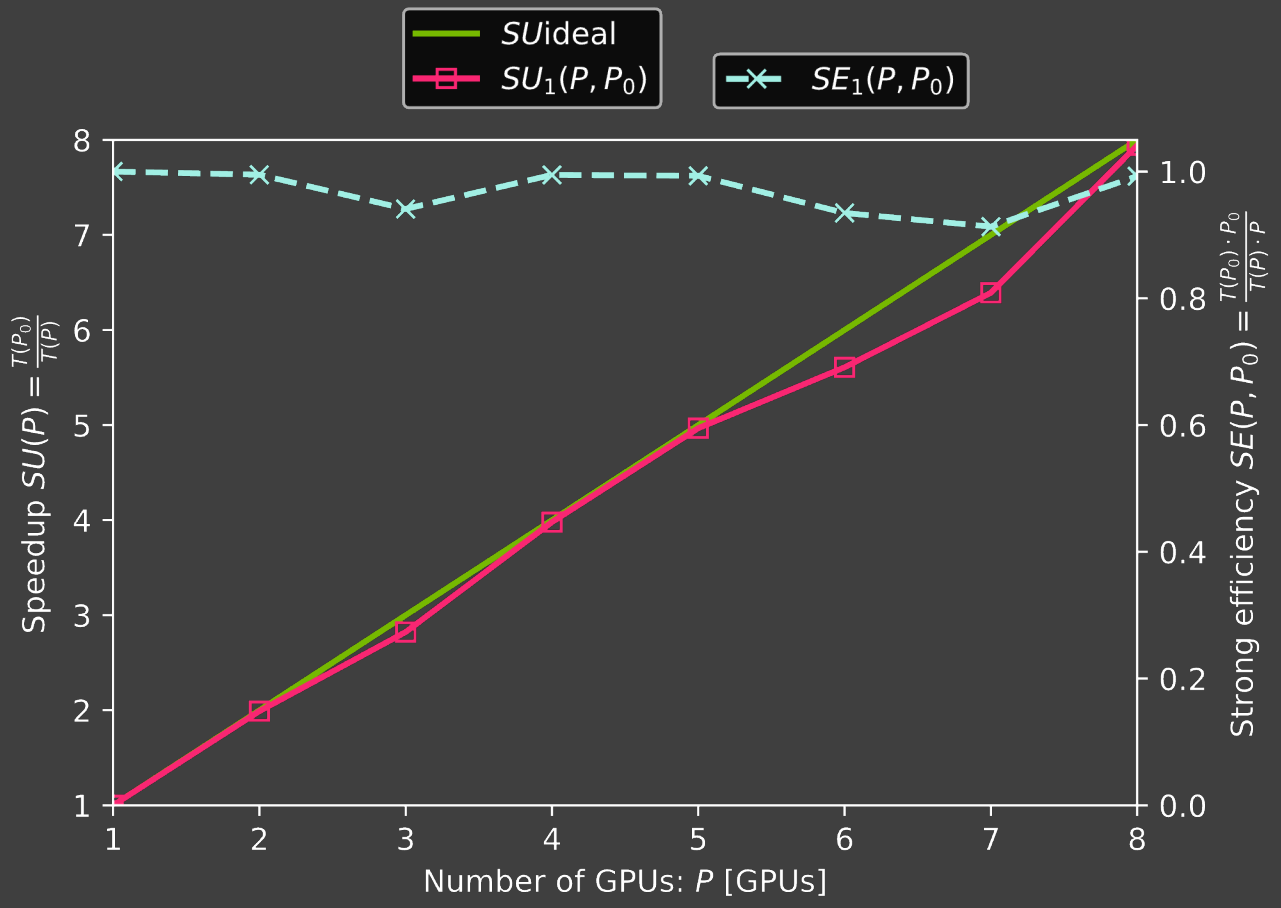
Maxwell Equations (DEMO)
Compute graphs

```
sender auto maxwell(scheduler auto& compute, scheduler auto& writer) {
  return
    repeat_n(n_output_iterations,
        repeat_n(n_inner_iterations,
            std::schedule(compute)
            | std::bulk(grid.cells, update_h(accessor))
            | halo_exchange(grid.border, hx, hy)
            | std::bulk(grid.cells, update_e(time, dt, accessor))
            | halo_exchange(grid.border, hx, hy)
        )
        | std::transfer(writer)
        | std::then(vtk_write(accessor))
    )
    | std::then([] { printer{"simulation complete"}; });
}

auto work = maxwell(cuda::execution::distributed_scheduler,
                    inline_scheduler);
std::this_thread::sync_wait(work);
```



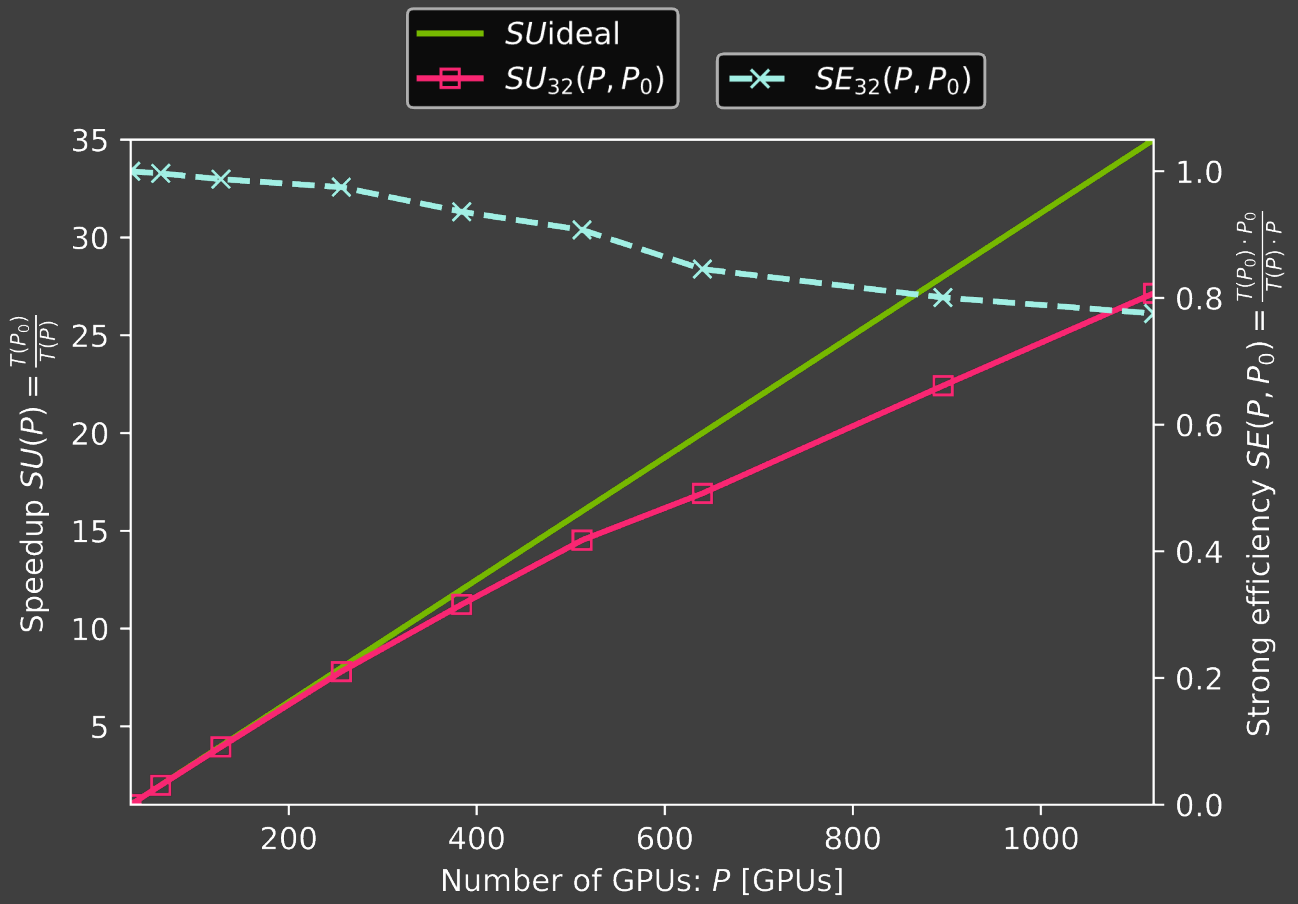
Maxwell Equations (DEMO)
DGX-A100 640 Strong scaling

- 8x NVIDIA A100-SXM4-80 GPUs (2 TB/s BW)
- Fully-connected with NVIDIA NVLink3 and NVSwitch
- 10x NVIDIA Mellanox Connect-X 6 NICs
- 2x AMD EPYC 7742 CPUs



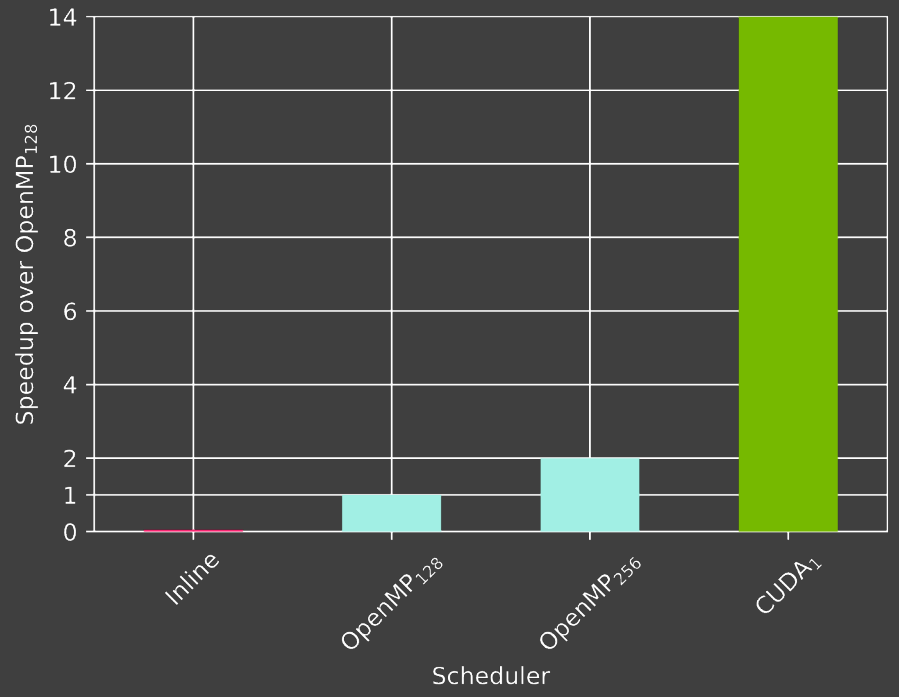
Maxwell Equations (DEMO)
Selene SuperPOD Strong scaling

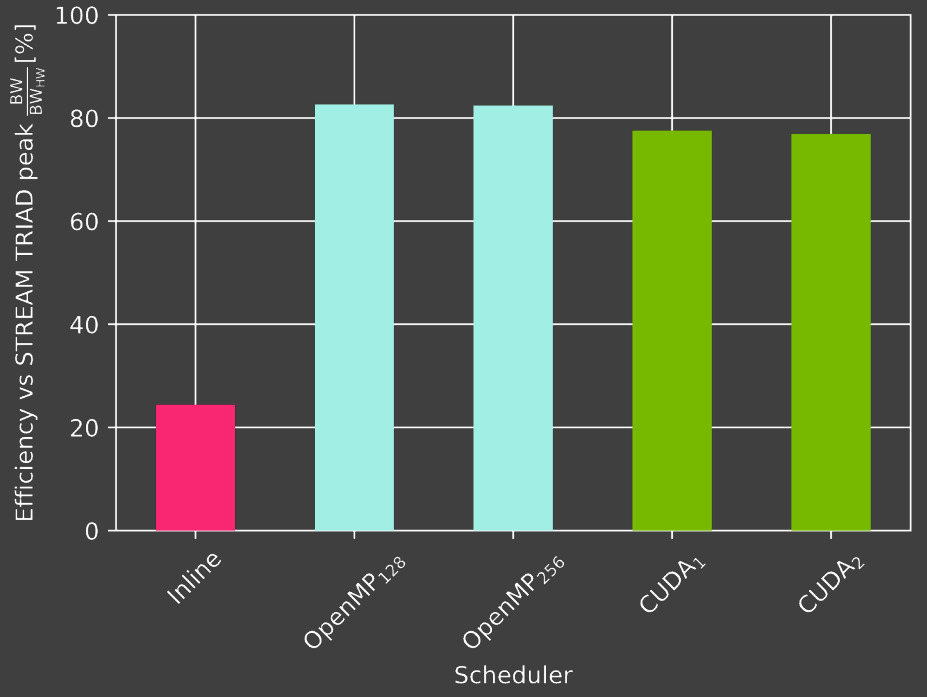
- #6 Top500 | #10 Green500 | #1 MLPerf | #5 HPCG
- 4x NVIDIA SuperPODs full fat-tree connected
- 560x NVIDIA DGX-A100 640 (4x 140)
- 4480x NVIDIA A100-SXM4-80 GPUs



Maxwell Equations (DEMO)
Raw performance (%peak)

```
std::sync_wait(maxwell(inline_scheduler,           inline_scheduler));
std::sync_wait(maxwell(openmp_scheduler,           inline_scheduler));
std::sync_wait(maxwell(cuda::distributed_scheduler, inline_scheduler));
```

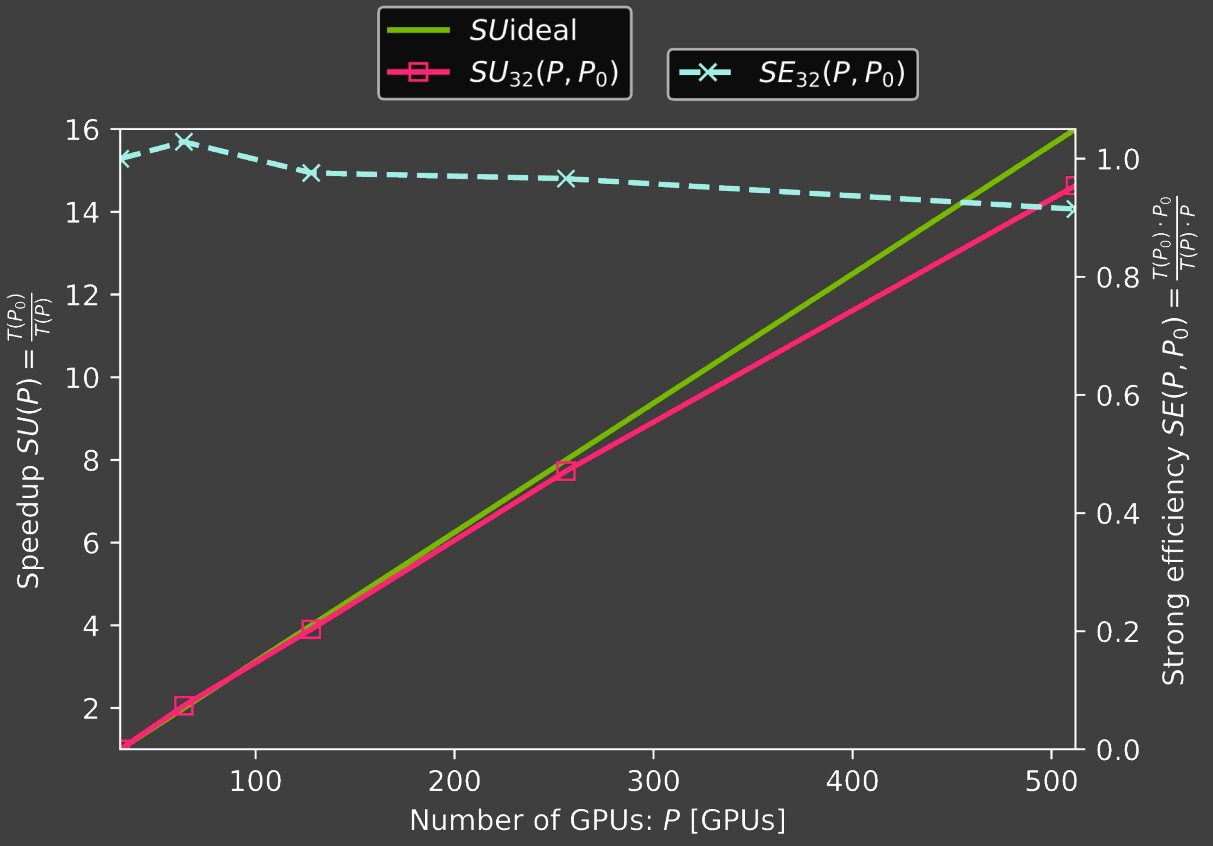


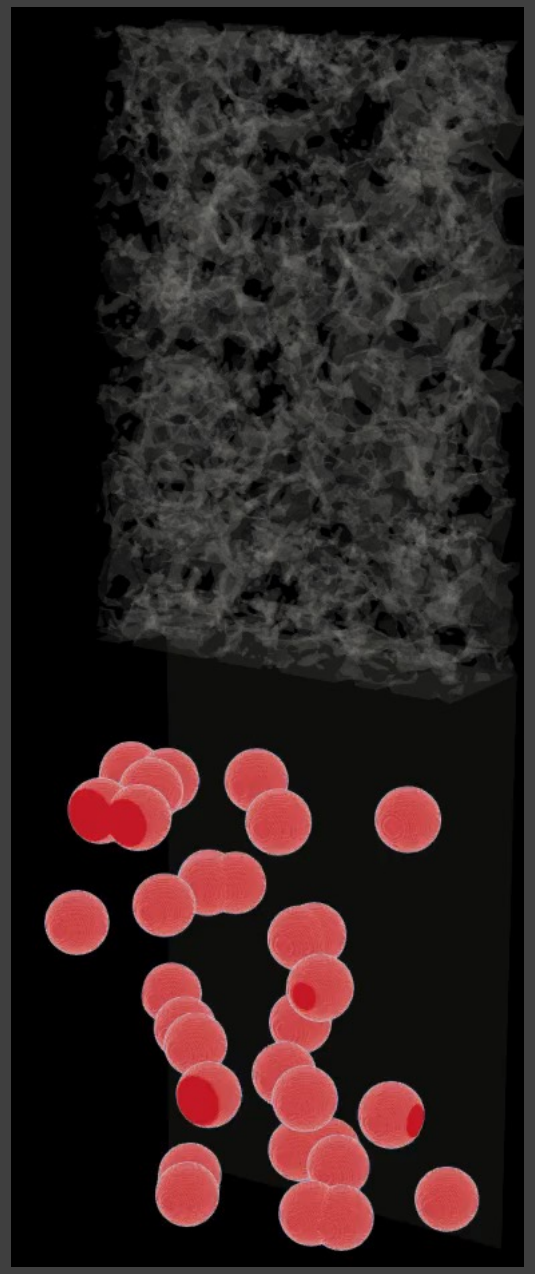


- CPUs: AMD EPYC 7742 CPUs, GPUs: NVIDIA A100-SXM4-80
- Inline (1 CPU HW thread), OpenMP-128 (1x CPU), OpenMP-256 (2x CPUs), Graph (1x GPU), Multi-GPU (2x GPUs)
- clang-12 with -O3 -DNDEBUG -mtune=native -fopenmp

Palabos Carbon Sequestration

Multi-phase flow through porous media and S&R





- Jonas Latt (University of Geneva), Christian Huber (Brown University), Georgy Evtushenko (NVIDIA), Gonzalo Brito (NVIDIA)

Maxwell Equations (DEMO)
Want to learn more?

- Eric Niebler, *Working with Asynchrony Generically: A Tour of C++ Executors*, Part 1 and Part 2, CppCon'21
- P2300: std::execution
- NVIDIA implementation of P2300 (GitHub)
- Eric Niebler, A Unifying Abstraction for Async in C++, CppCon'19

References

References: parallel algorithms

CppCon talks:

- Thomas Rodgers, *Bringing C++ 17 Parallel Algorithms to a standard library near you*, 2018
- Olivier Giroux, *Designing (New) C++ Hardware*, 2017
- Dietmar Kühl, *C++17 Parallel Algorithms*, 2017
- Bryce Adelstein Lelbach, *The C++17 Parallel Algorithms Library and Beyond*, 2016

GTC talks (available at https://nvidia.com/nvidia/gtc):

- Simon McIntosh-Smith et al., *How to Develop Performance Portable Codes using the Latest Parallel Programming Standards*, Spring 2022
- Jonas Latt, *Porting a Scientific Application to GPU Using C++ Standard Parallelism*, Fall 2021
- Jonas Latt, *Fluid Dynamics on GPUs with C++ Parallel Algorithms: State-of-the-Art Performance through a Hardware-Agnostic Approach*, Spring 2021

Software:

- NVIDIA C++ Standard Library Parallel algorithms: https://github.com/nvidia/thrust
- NVIDIA C++ Standard Library: https://github.com/NVIDIA/libcudacxx

References: ranges and views

References

- Tristan Brindle, An Overview of Standard Ranges, CppCon 2019
- Eric Niebler, Ranges for the Standard Library, CppCon 2015
- Andrei Alexandrescu, Iterators Must Go, BoostCon 2009

Software

- Range-v3: https://github.com/ericniebler/range-v3
- All the ranges that didn't make it into C++20: https://github.com/TartanLlama/ranges

References: P2300 std::execution

Proposals

- P2300 Senders & Receivers

Implementations:

- Reference: https://github.com/brycelelbach/wg21_p2300_std_execution/
- Production: https://github.com/facebookexperimental/libunifex
- Others: https://github.com/dietmarkuehl/kuhllib

Talks

- Working with Asynchrony Generically: A Tour of C++ Executors Part I & Part II, CppCon'21
- A Unifying Abstraction for Async in C++, CppCon'19

References: P2300 std::execution

Proposals

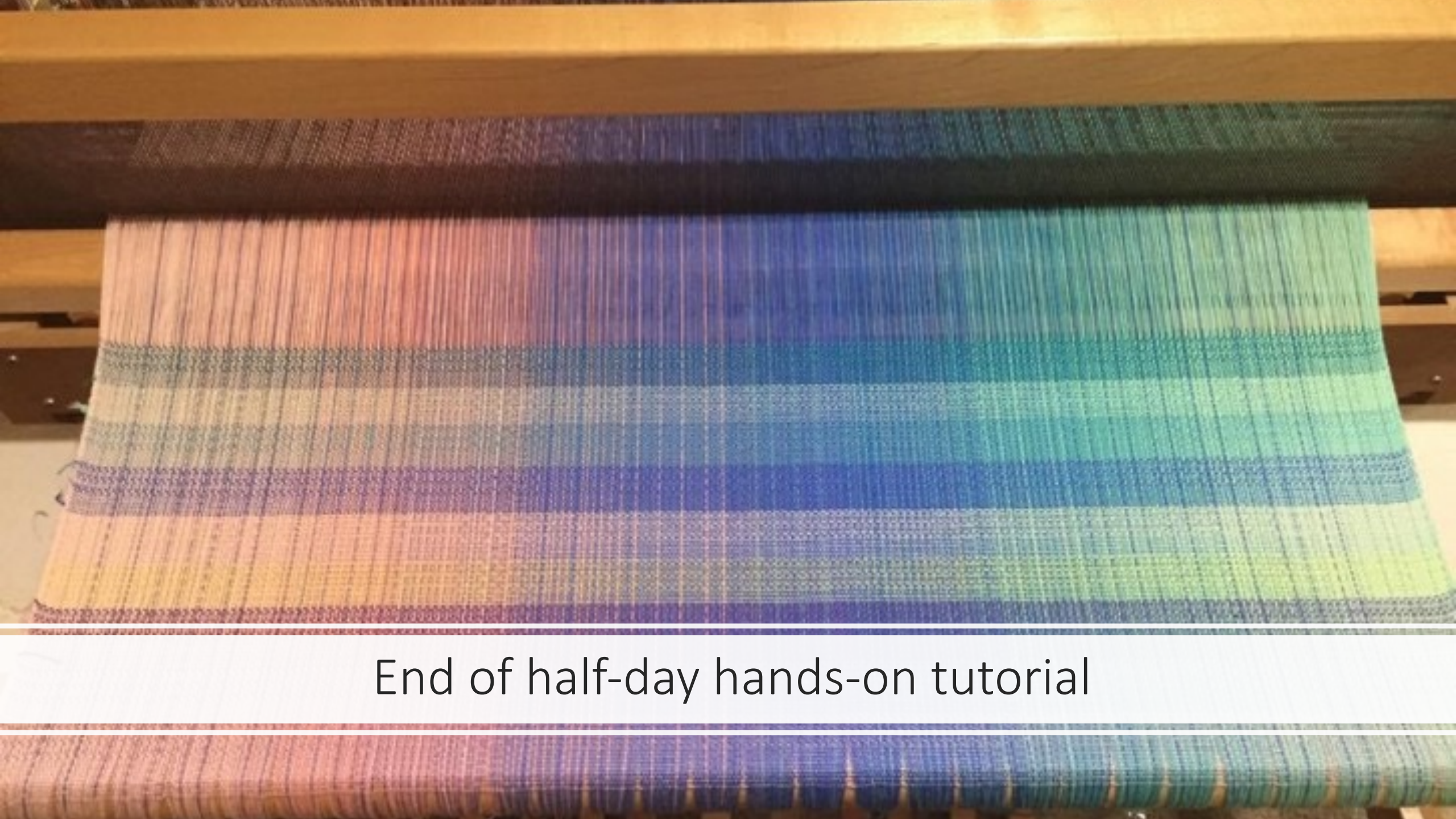
- P1637 A free function linear algebra interface based on the BLAS

Implementations

- Reference: https://github.com/kokkos/stdBLAS/

LEWG presentation:

- https://github.com/kokkos/stdBLAS/blob/main/lewg-presentation.md



End of half-day hands-on tutorial

Save your work!

If you want to preserve your modified exercises, these are stored to:

/lustre/workspaces/$USER/workspace-stdpar

Copy them after them course to your local machine using scp!

AGENDA: part II

Introduction to C++ Concurrency Primitives

- Threads, Atomics, Barriers, Mutexes

Lab 2: 2D heat equation (Part II)

- Overlapping communication behind computation

Atomic memory operations

- Memory Orderings, Acquire/Release semantics
- C++ primitives: atomic, atomic_ref, atomic_flag, fences
- Critical sections & starvation-free algorithms

Execution Policies & Element Access Functions

Lab 3: Parallel Tree Construction