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)

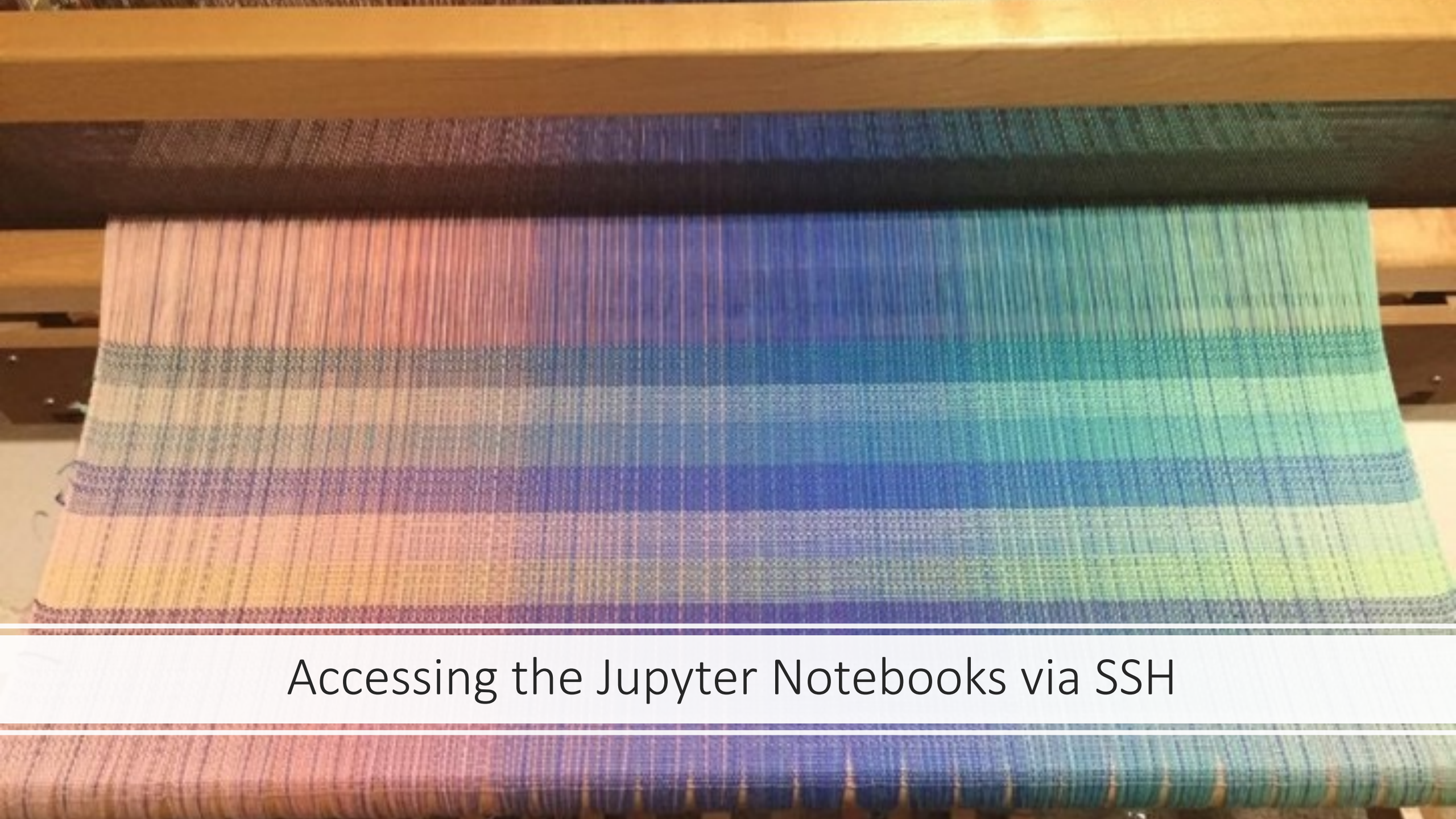
- Overlapping communication behind computation

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

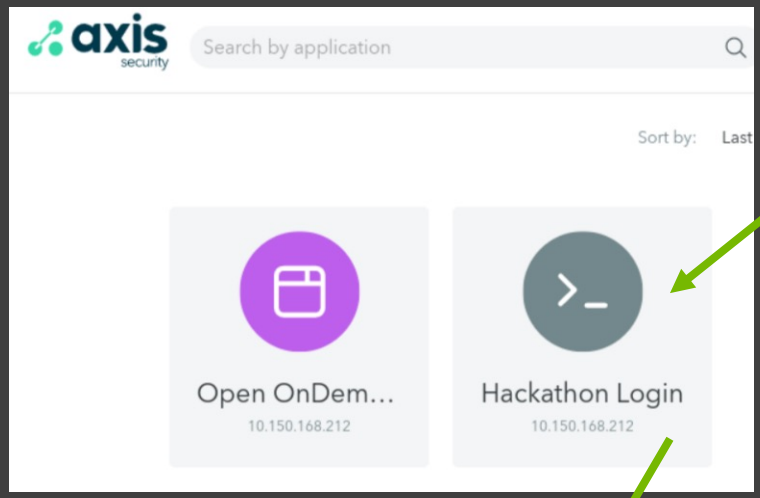
Lab 3: Parallel Tree Construction

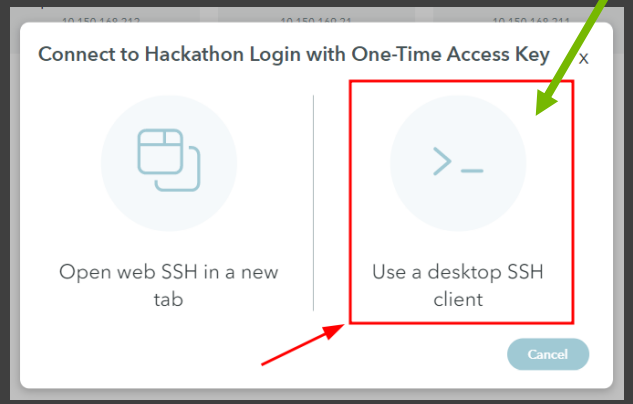


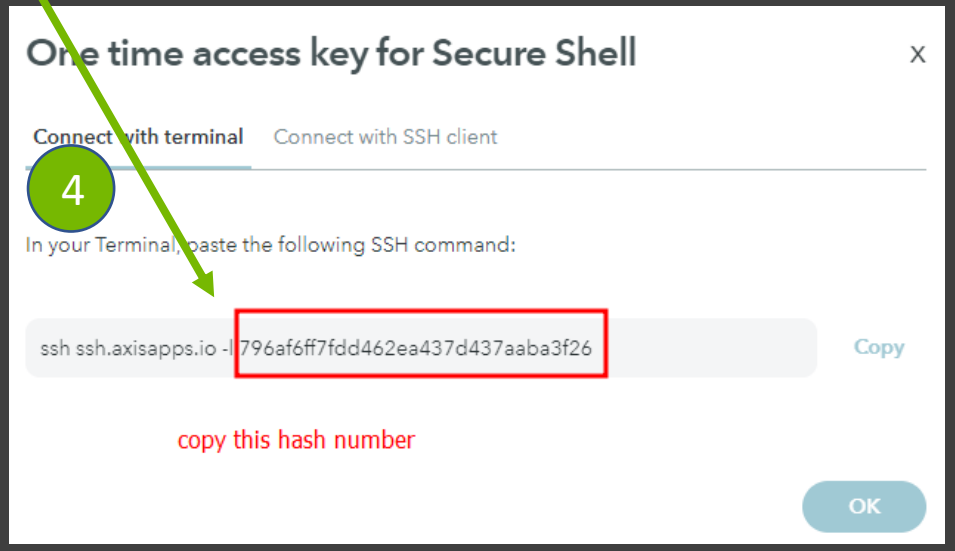
Accessing the Jupyter Notebooks via SSH

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

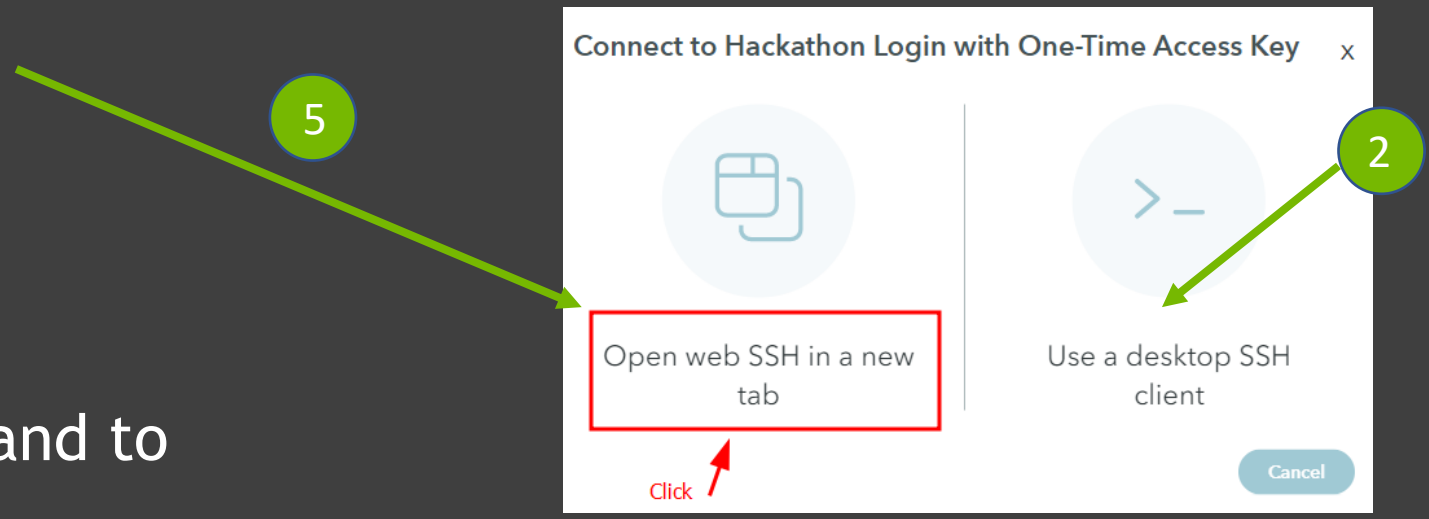






Start the Notebook job

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 hash 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)

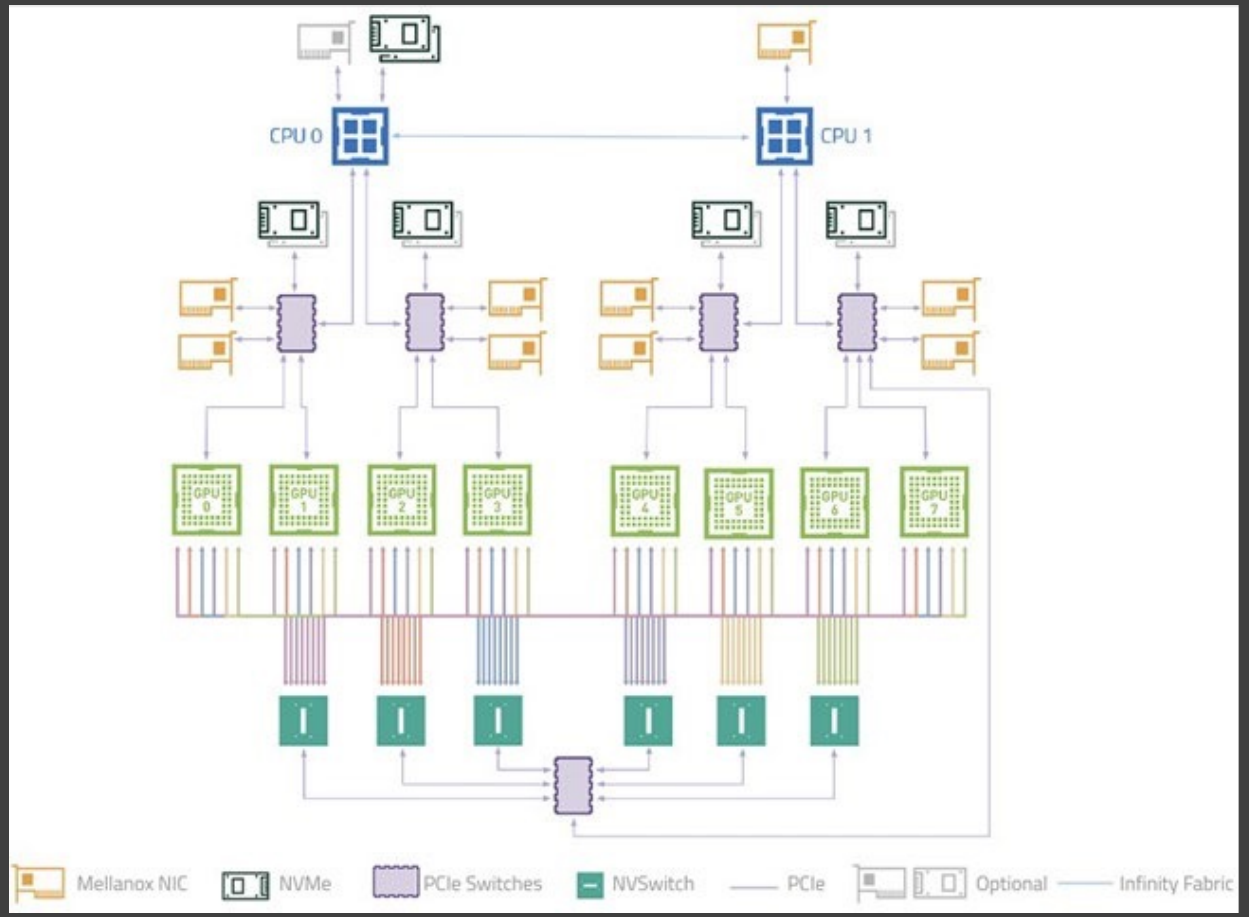
Cluster nodes: DGX-A100

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:

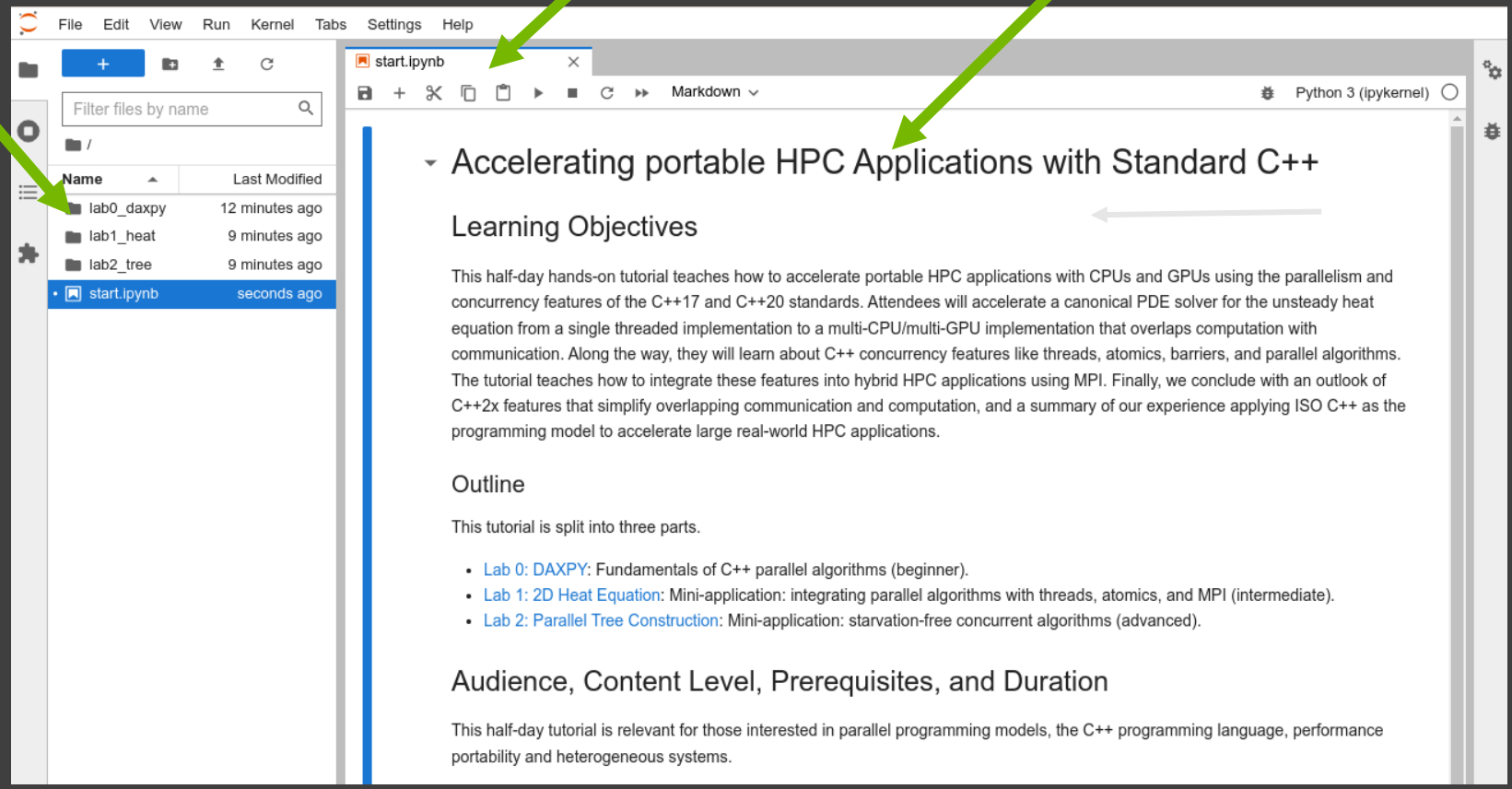
/lustre/workspaces/$USER/workspace-stdpar

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

Using the Jupyer Notebook

File Explorer:

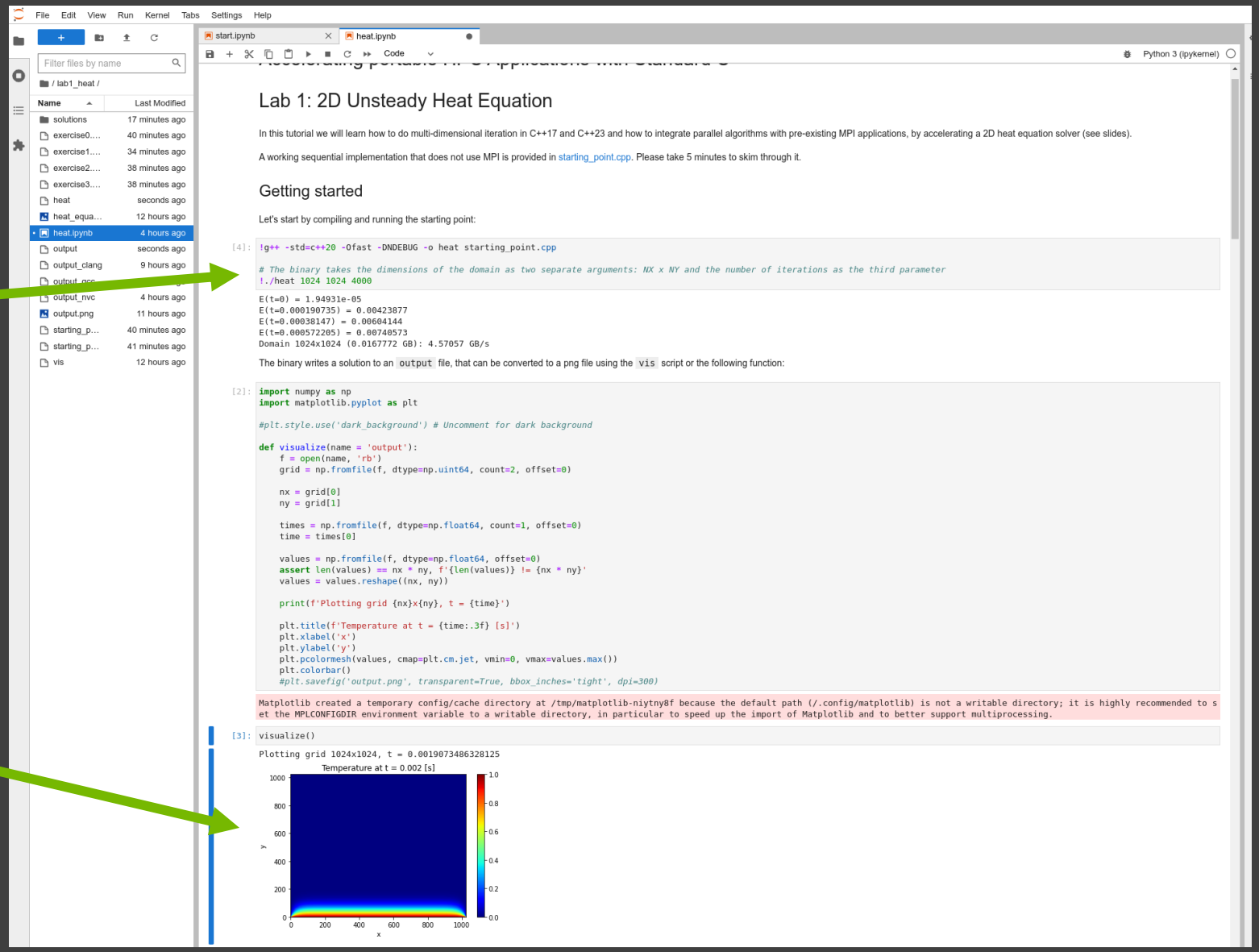
- Create
- Open
- Navigate



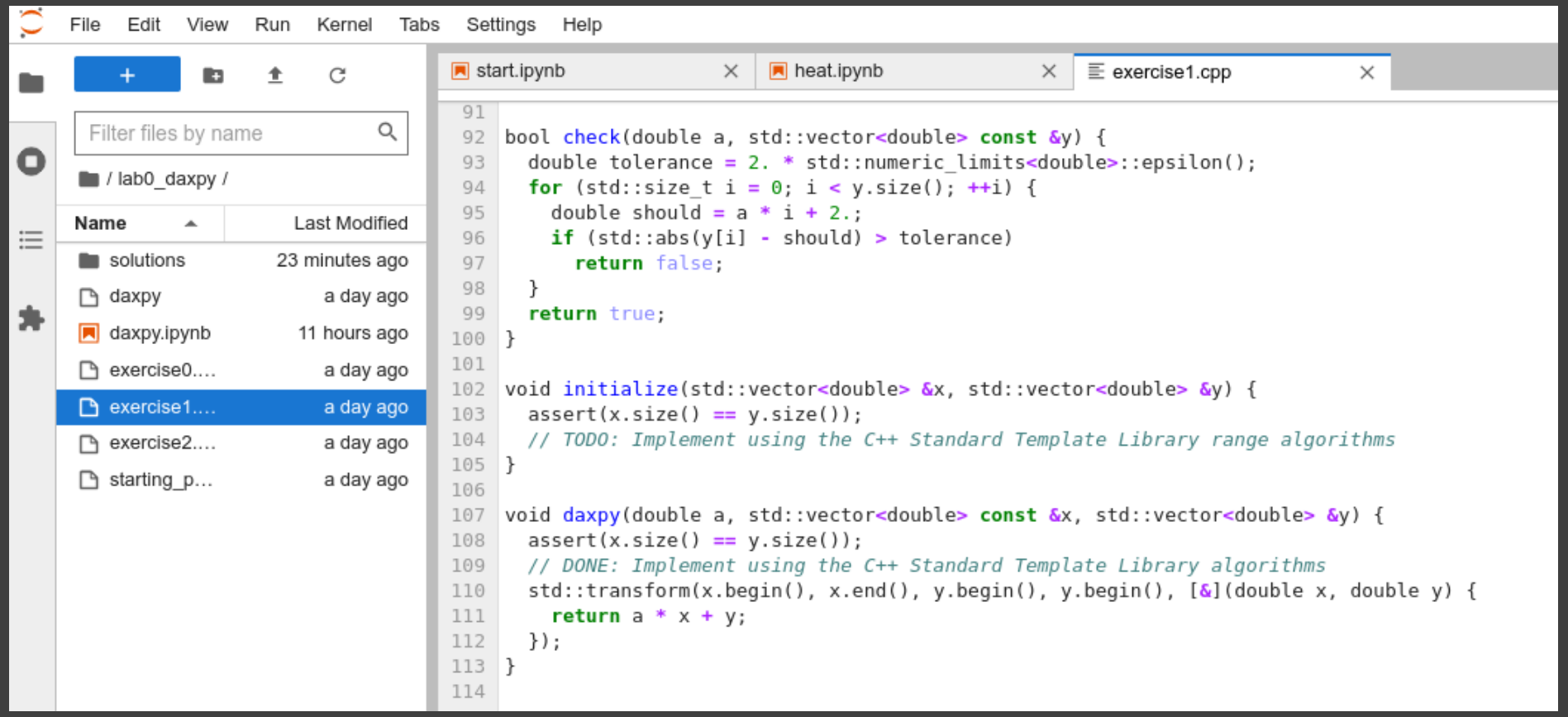
Labs

Run commands in code cells with CTRL+ENTER

Visualize the outputs of the simulations while learning

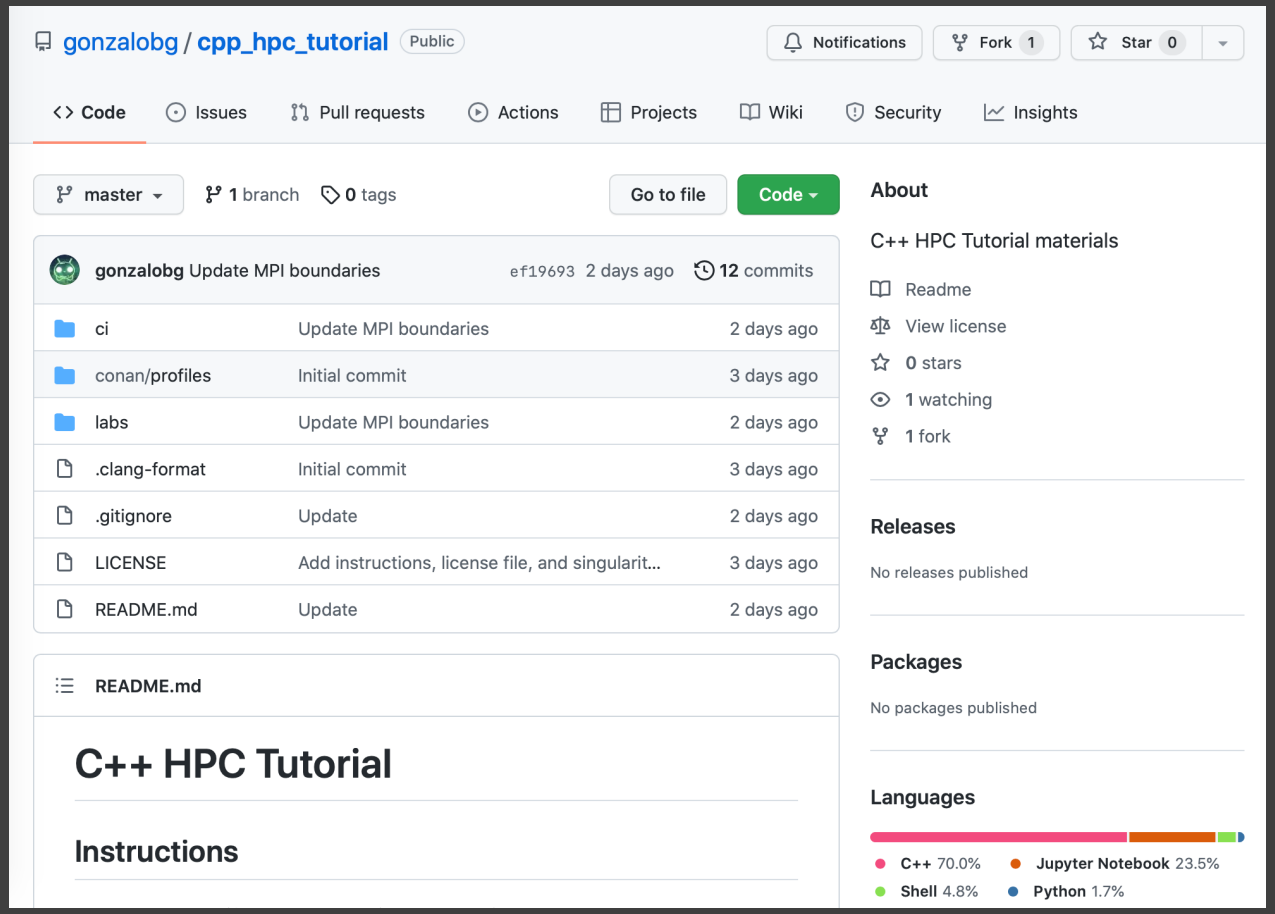


Labs: edit code in the browser



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:

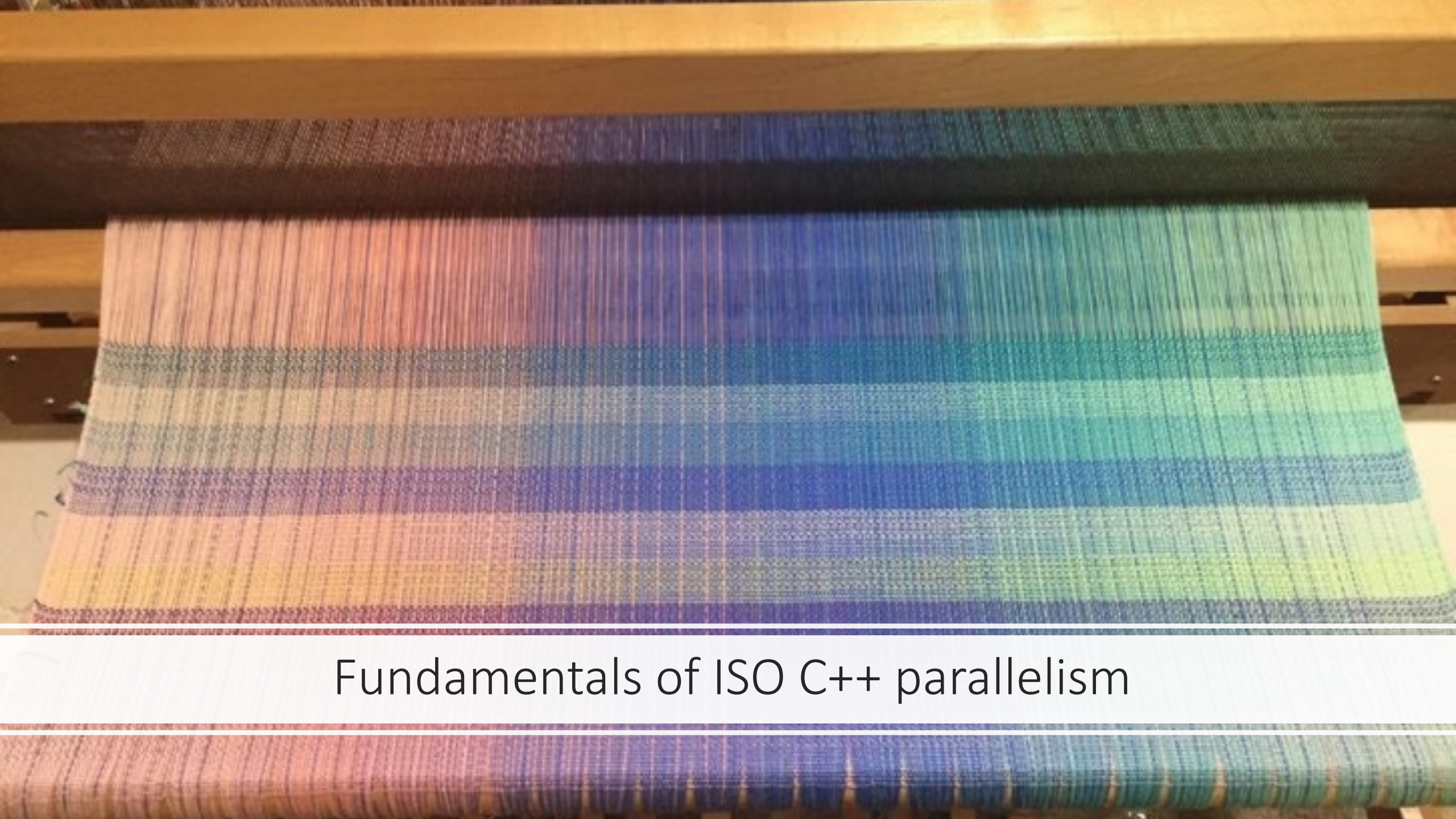
- [&,s] captures s "by value" (makes a copy) and all other used variables "by reference" (store pointers)
- [=,&v] captures v "by reference" (stores a pointer to v) and all other used variables "by value" (copy them)

```
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

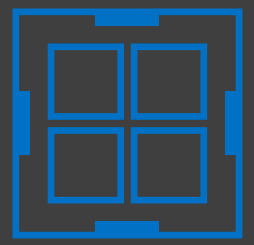
```
std::vector<double> v = {1, 2, 3, 4}, w(4);
std::transform(std::execution::par, begin(v), end(v), begin(w),
               [](const double& el) { return 2. * el; });
```

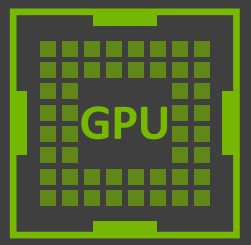
ISO C++ parallel algorithms
Compiler selects target for parallel execution

```
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

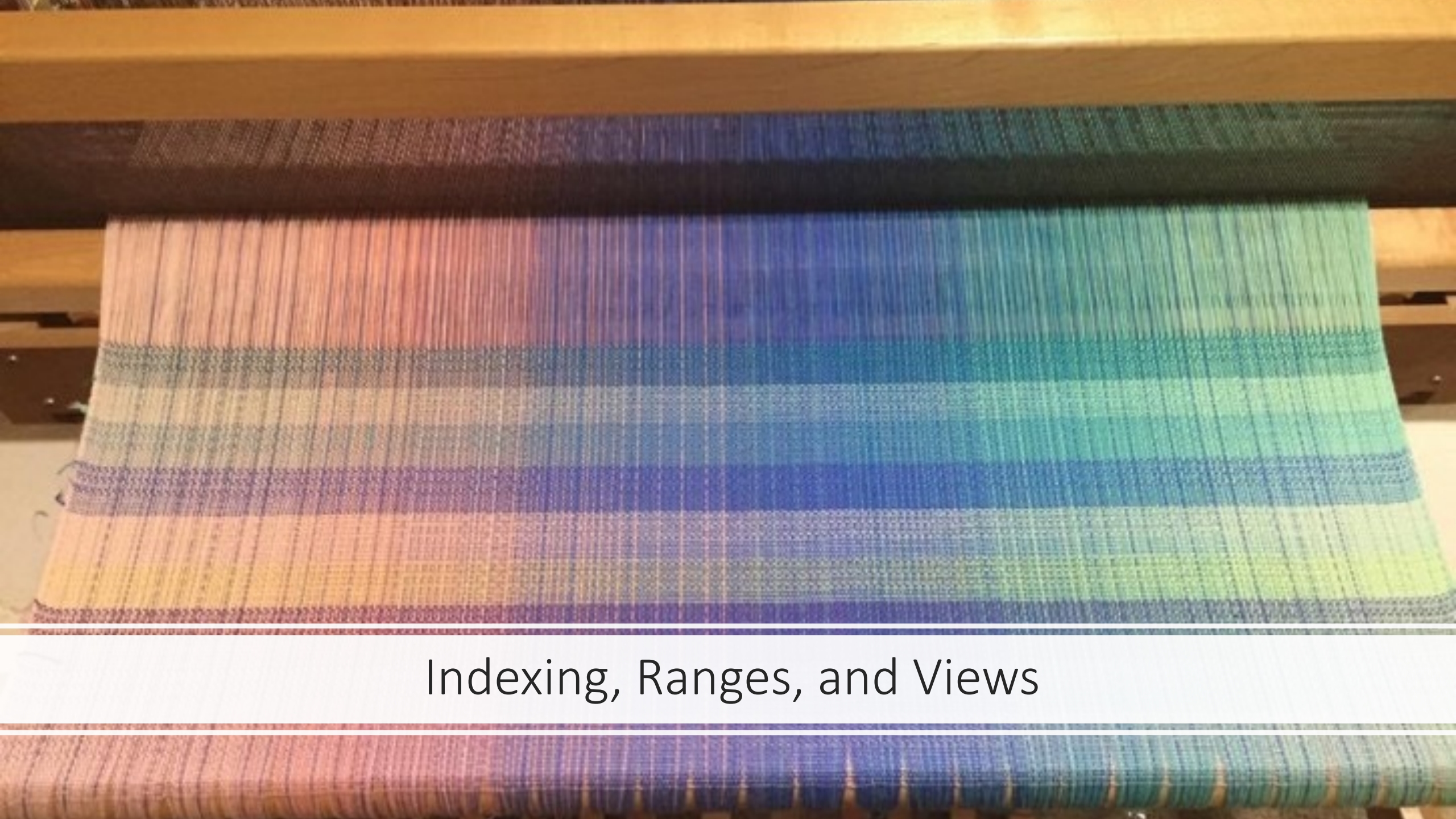
std::remove, std::remove_if
std::replace, std::replace_if, std::replace_copy, std::replace_copy_if
std::unique / std::unique_copy

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…

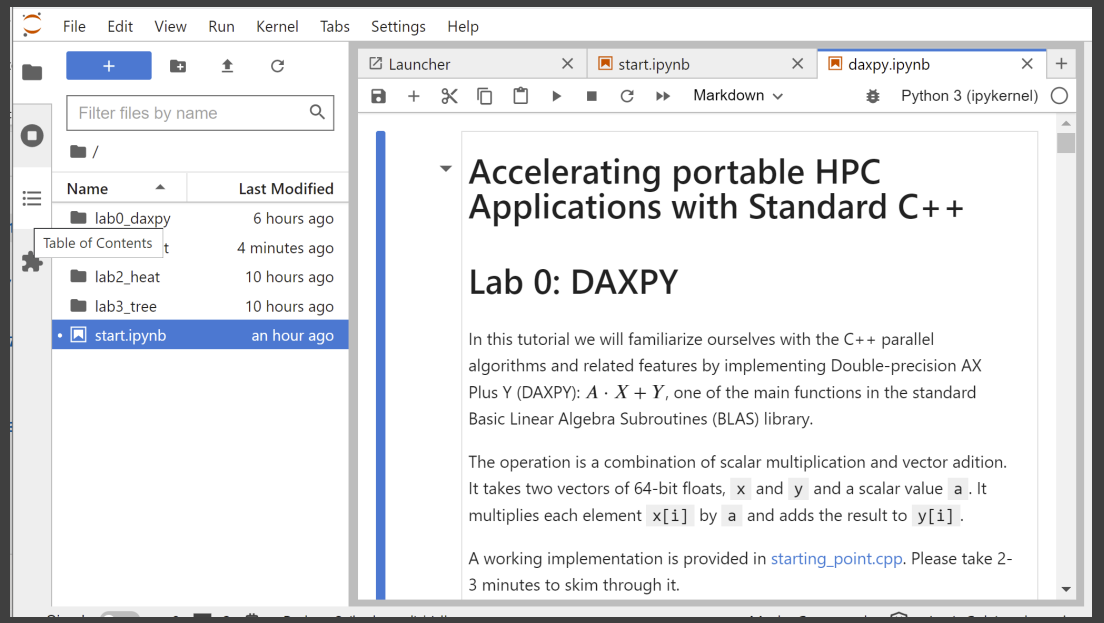
```
for (int i = 0; i < N; ++i) {
    y[i] += a * x[i];
}
```

Lab 1: BLAS DAXPY
3 Exercises

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

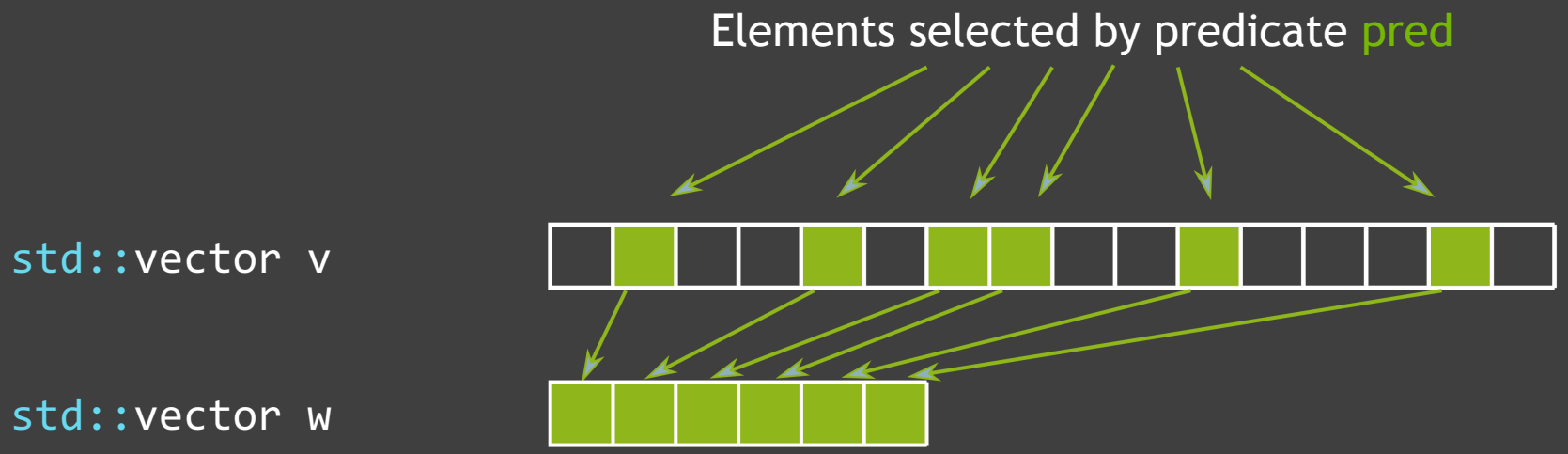
- **Exercise 2:** rewrite the for-loop implementation of the "initialization" kernel using sequential STL algorithms with any of the indexing approaches discussed in the tutorial

- **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.

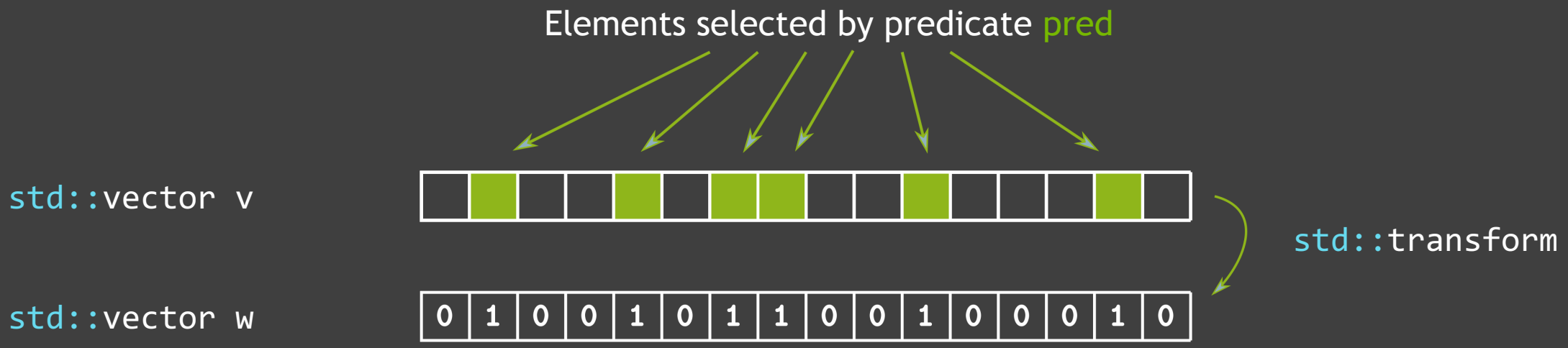


- Sequentially no problem, but how to write a parallel algorithm ?

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

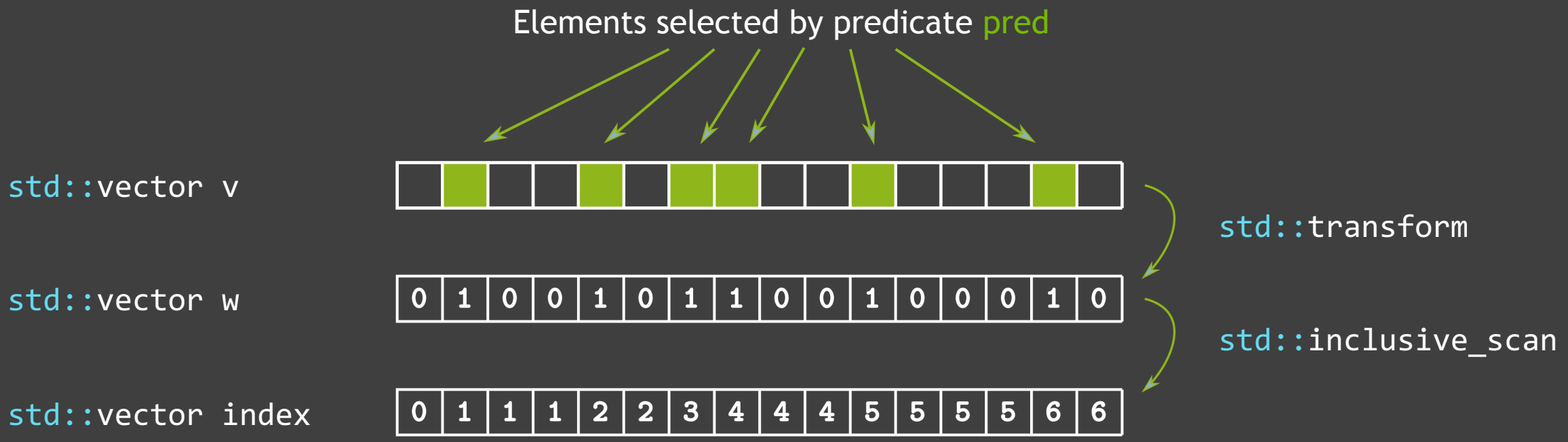
Lab 1: Select

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



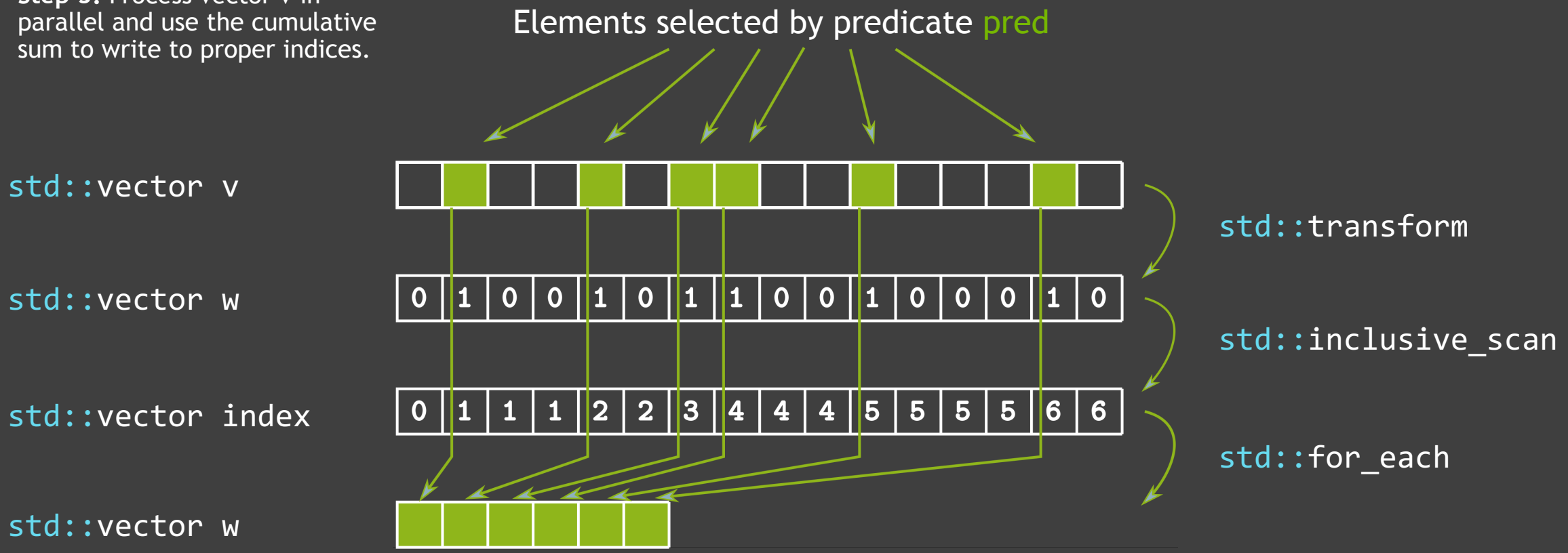
Lab 1: Select

- **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.



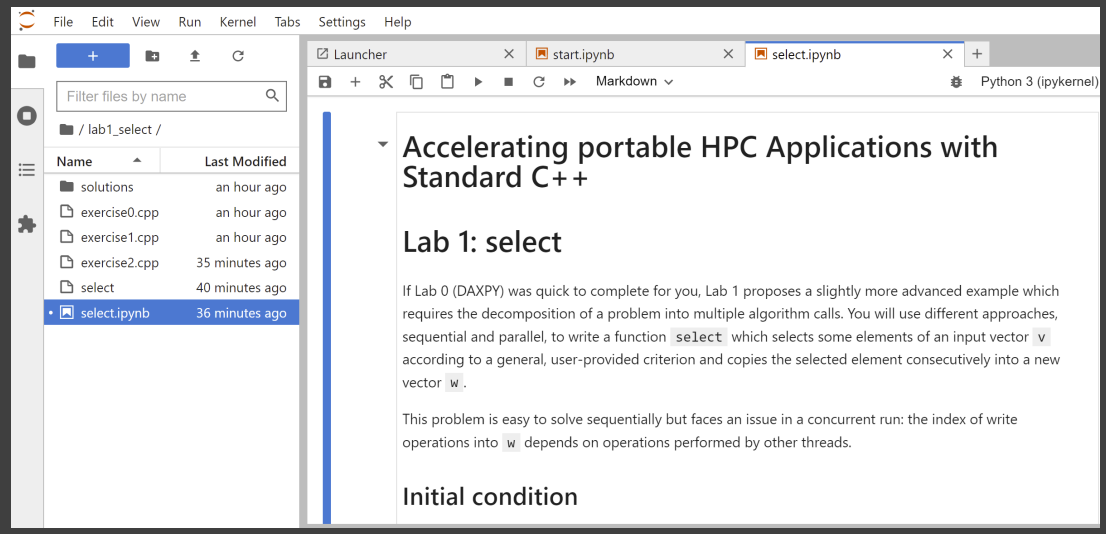
Lab 1: Select
3 Exercises

- **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.

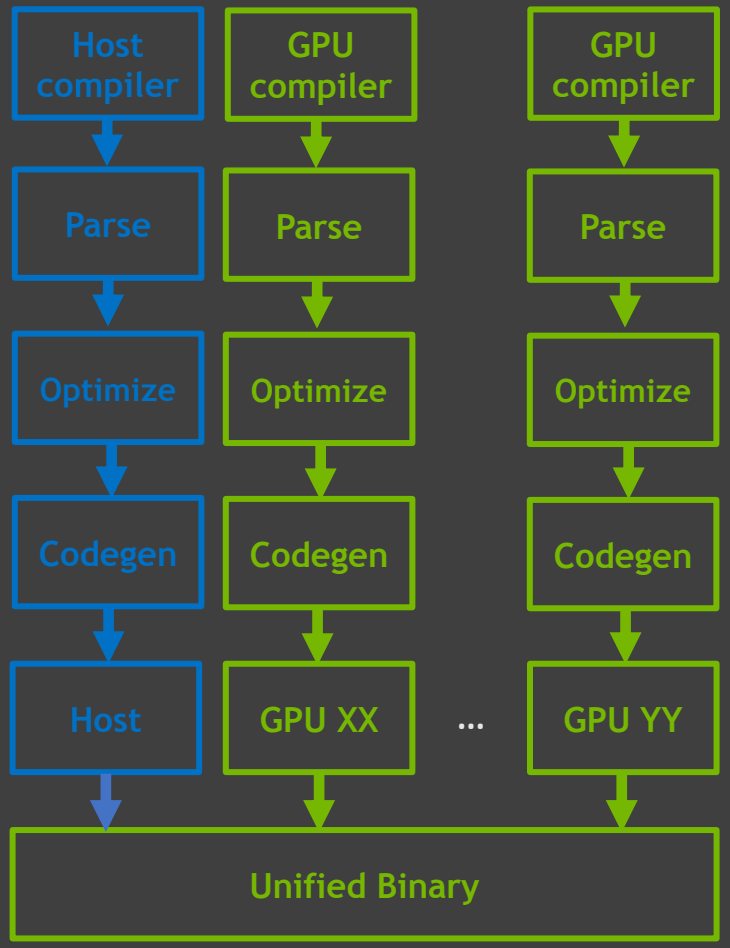


Lab 1: BLAS DAXPY Solutions (DEMO)

Lab 1: Select Solutions (DEMO)

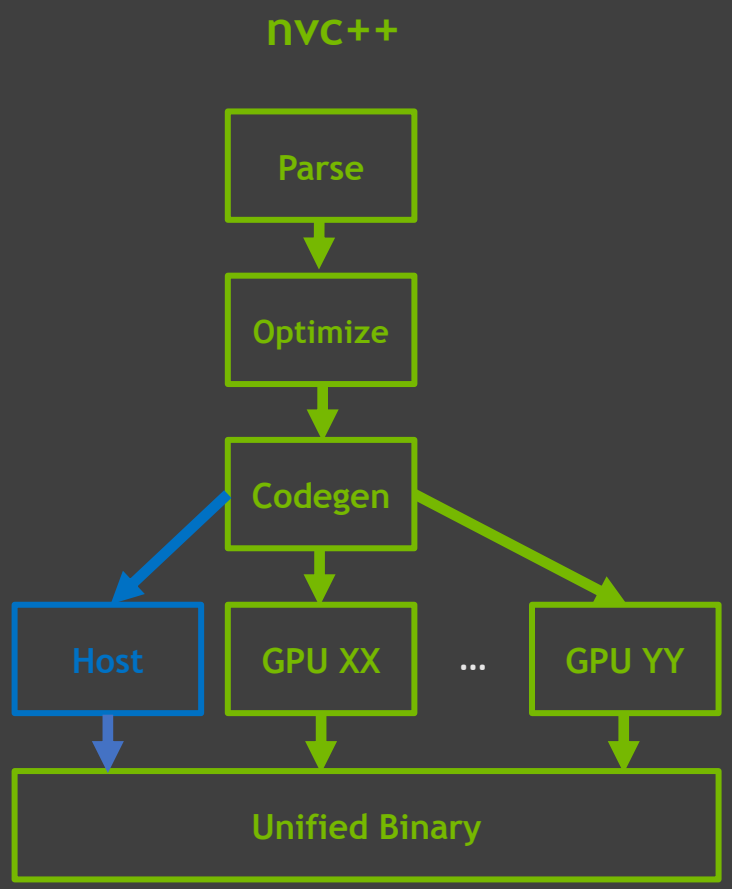
NVC++: A unified compiler for heterogeneous systems

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

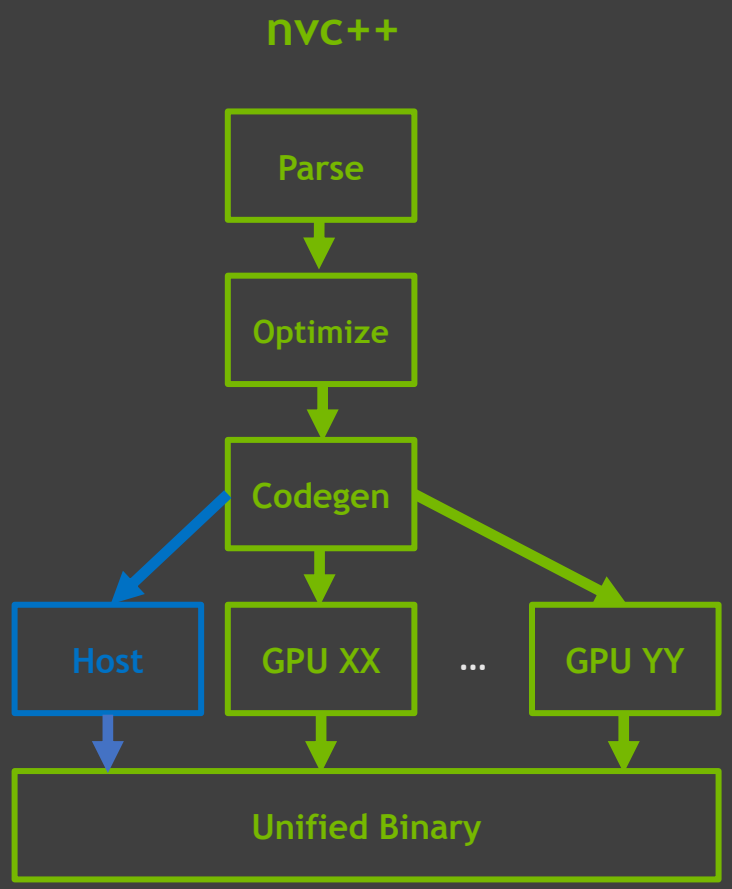
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- 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); });
```

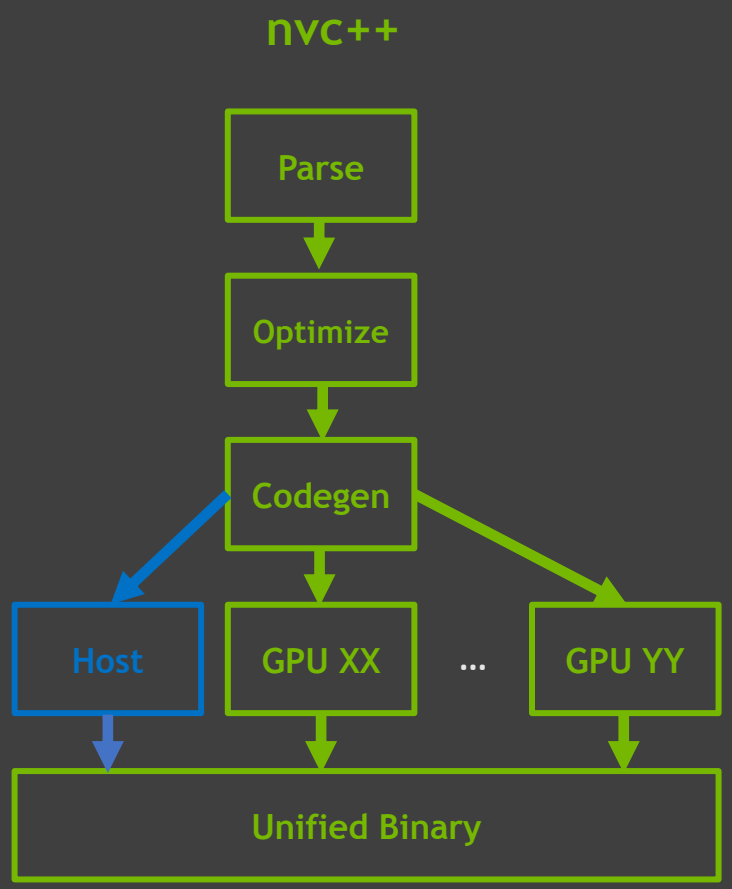
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- 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

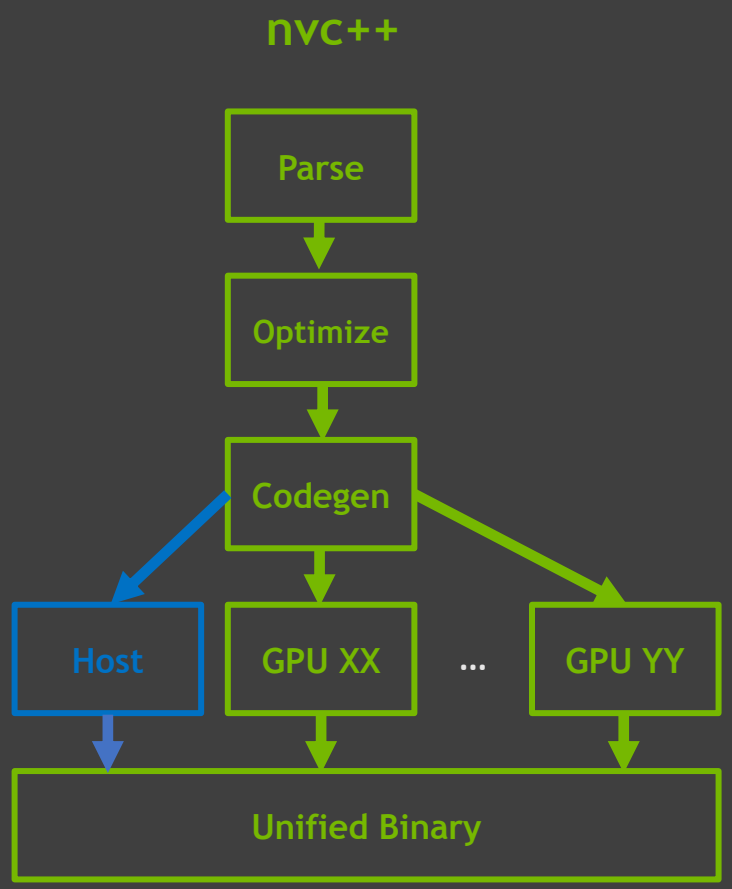


- Separately compile host and device code:

```
// 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));
});
```

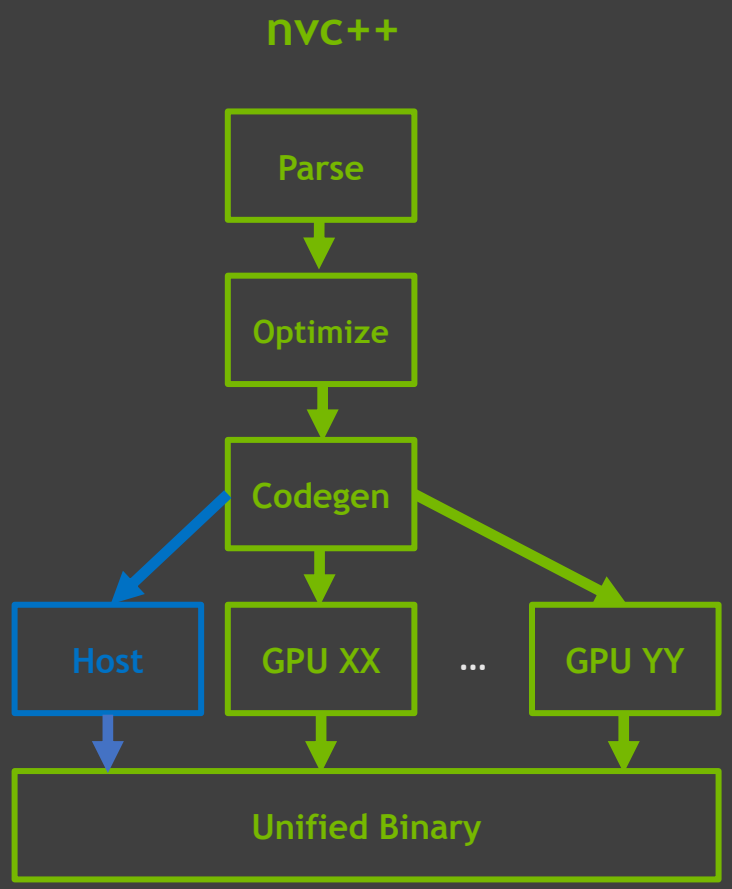
NVC++: NVIDIA C++ COMPILER
Unified Heterogeneous Compiler



- 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++.

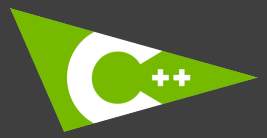
libcu++ (NVCC, NVC++)

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**

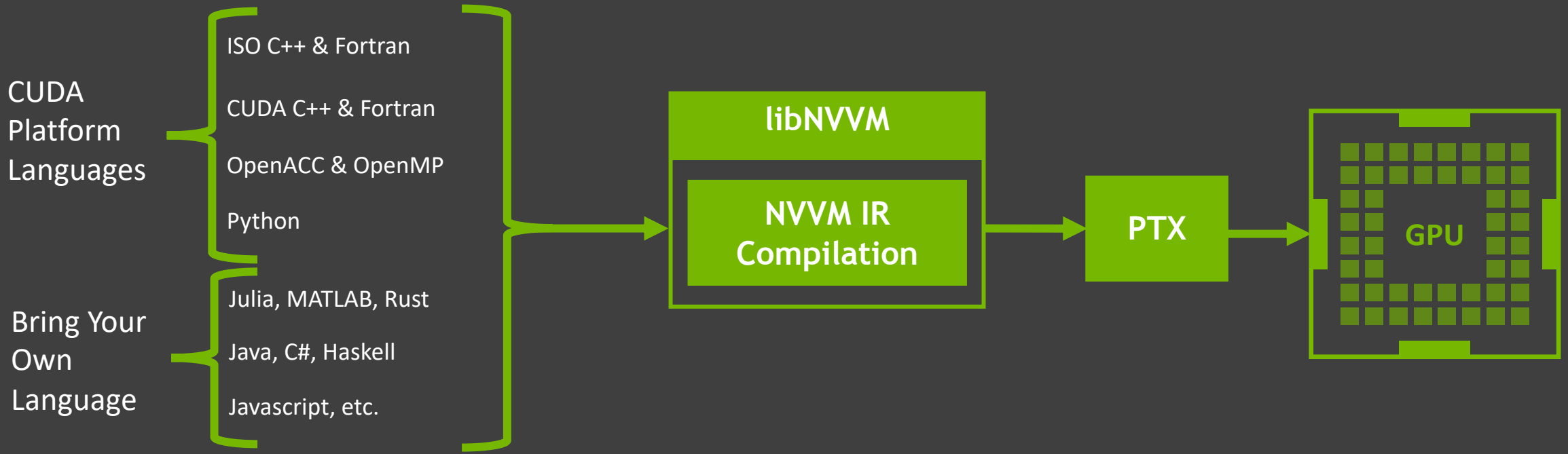
- Synchronization: atomic, atomic_ref, barrier, semaphores, …
- Time: chrono clocks, durations, rations, dates, calendars, …
- Utilities: type traits, array, tuple, complex, byte, …
- **Thread scopes:** thread, block, device, system
  - atomic<T, thread_scope>, barrier<thread_scope>, …
- CUDA extensions:
  - 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)

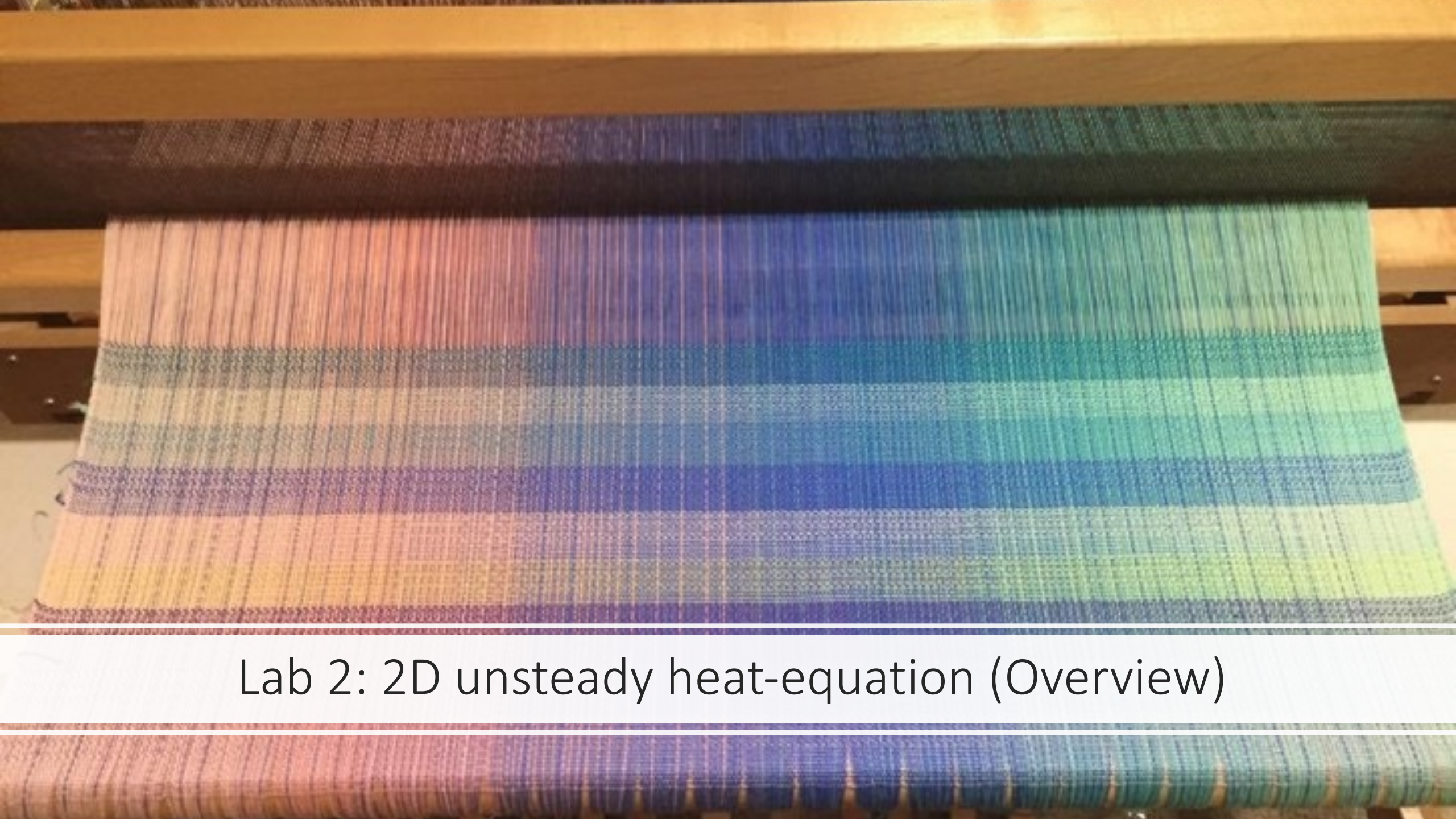
```
double GLOBAL;
void example(double a) {
    vector<double> x;
    auto ints = views::iota(0, x.size());
    std::for_each(par, ints.begin(), ints.end(),
       [x = x.data(), GLOBAL, a](int idx) {
            x[idx] += GLOBAL * a;
    });
}
```

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

NVC++: NVIDIA C++ COMPILER References

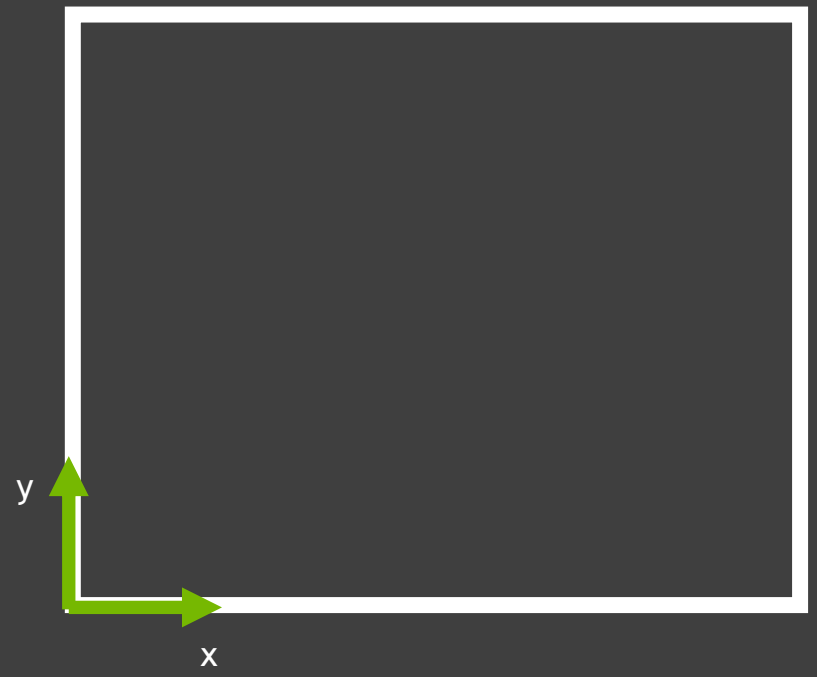
- NVC++ Parallel Algorithms Guidelines: https://docs.nvidia.com/hpc-sdk/compilers/c++-parallel-algorithms/index.html#guidelines

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

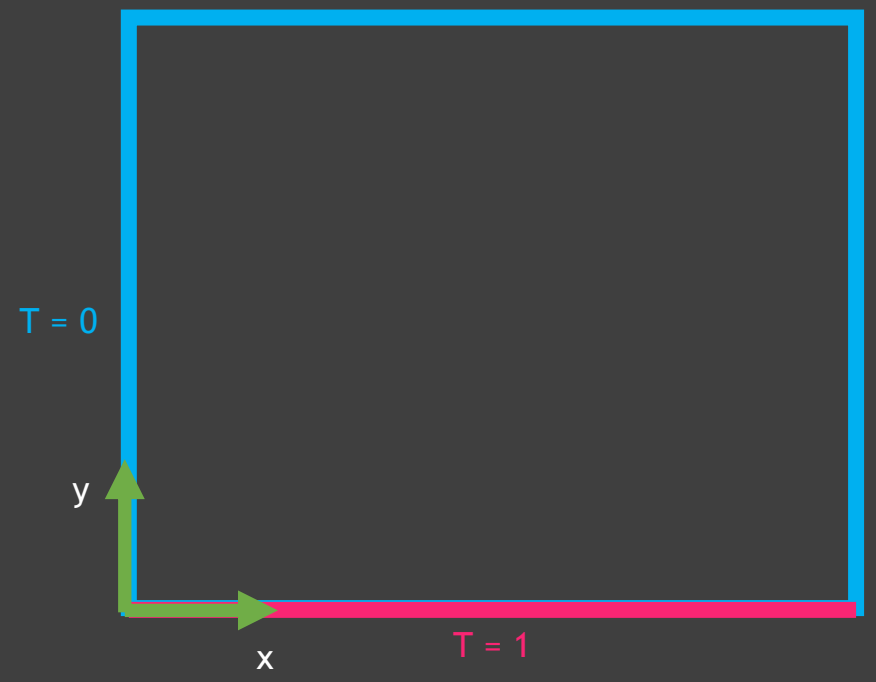


Lab 2: 2D unsteady heat-equation (Overview)

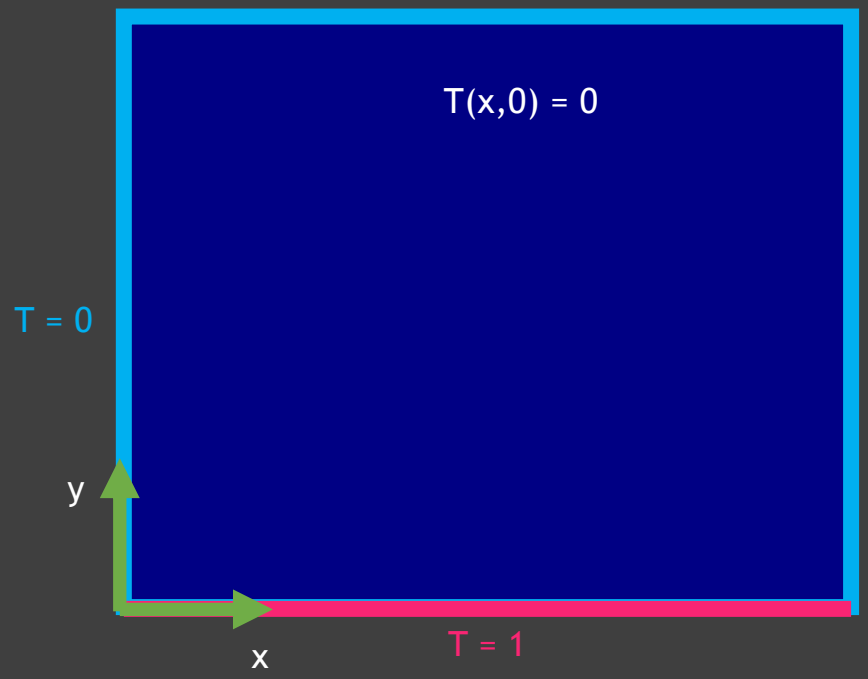
2D unsteady heat equation
Physical domain



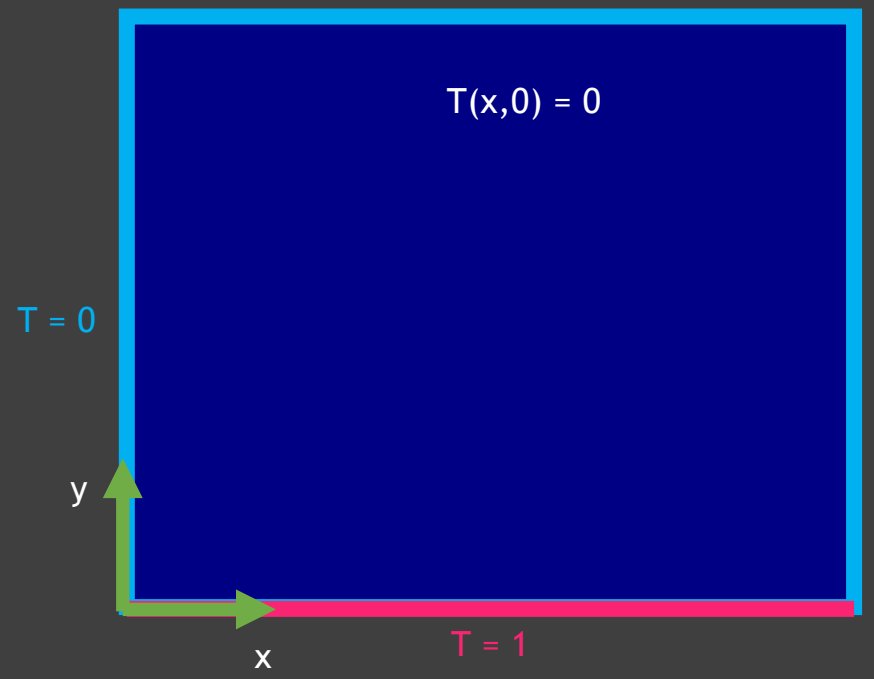
2D unsteady heat equation
Boundary conditions



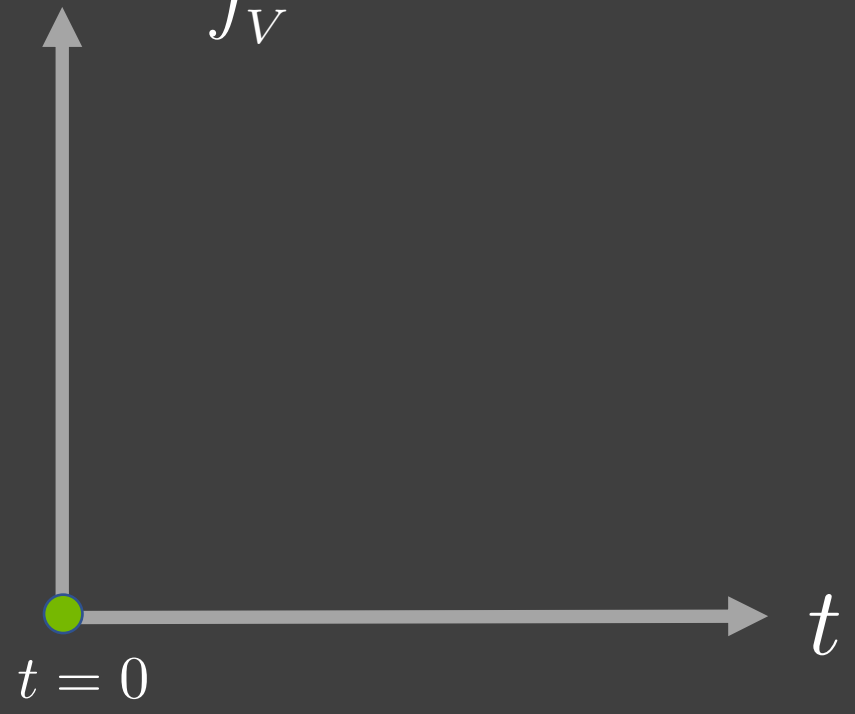
2D unsteady heat equation
Initial condition



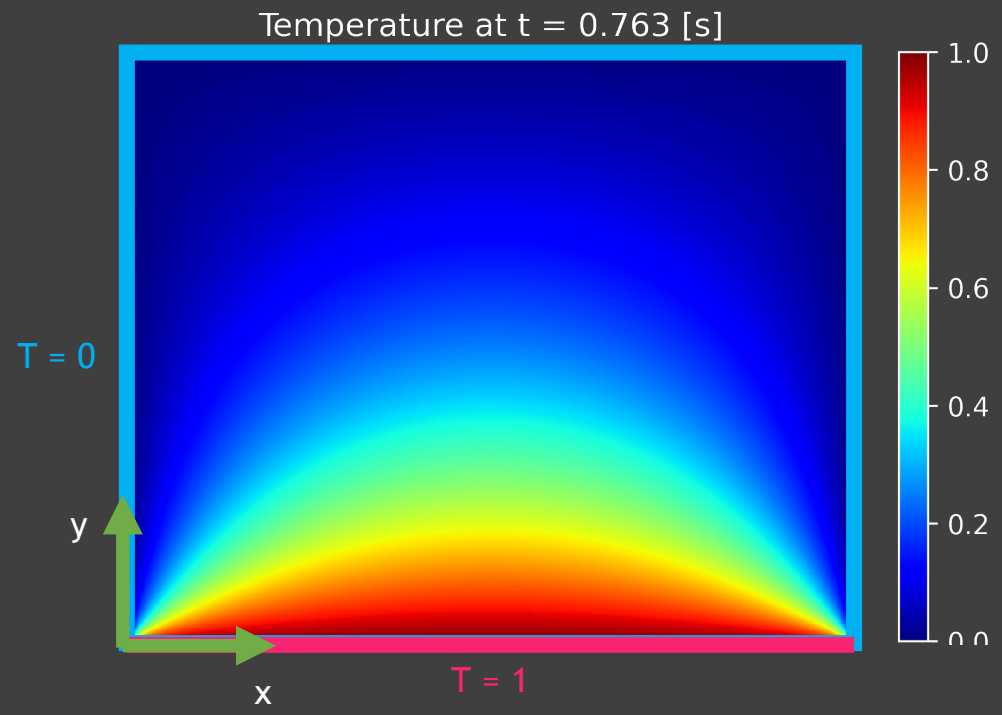
2D unsteady heat equation
Total thermal energy



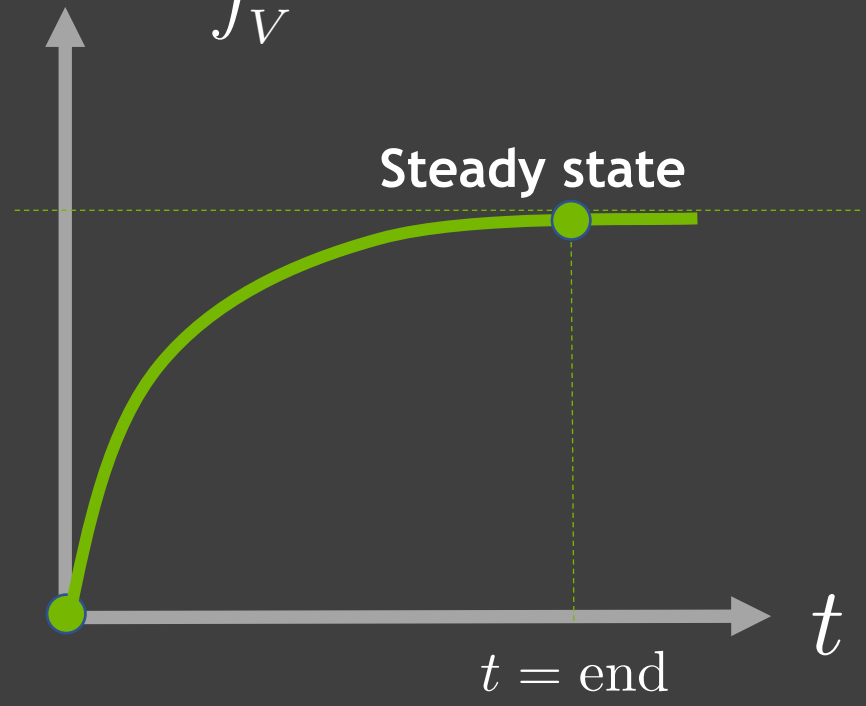
$$E(t) = \int_V \rho c T(x,t)\, dV$$



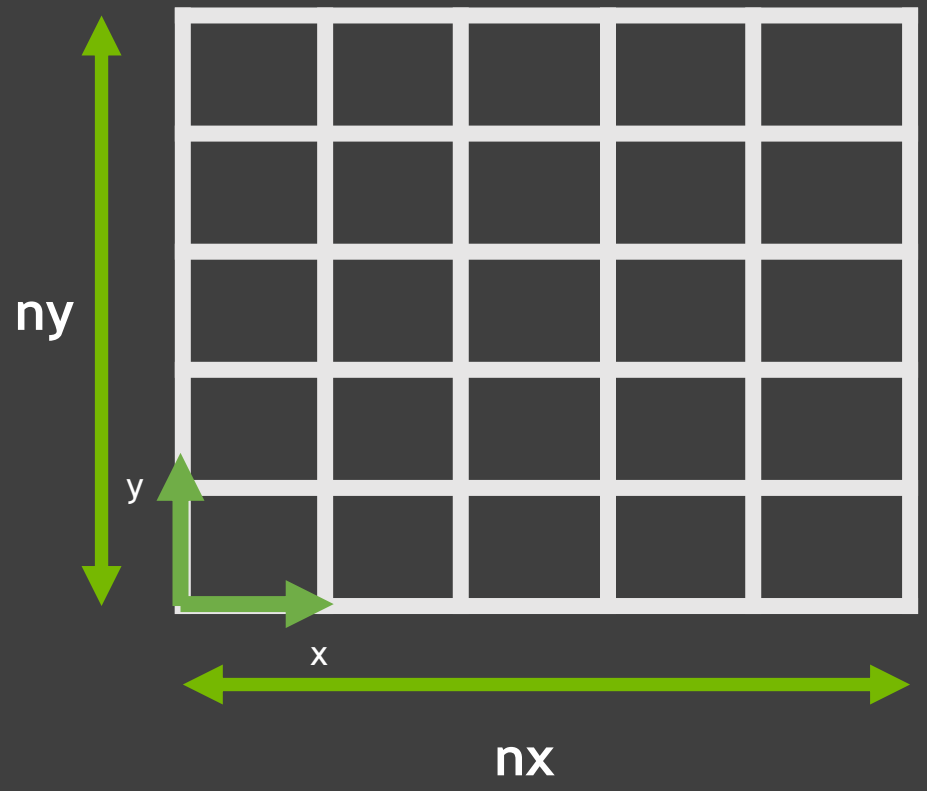
2D unsteady heat equation
Steady state solution



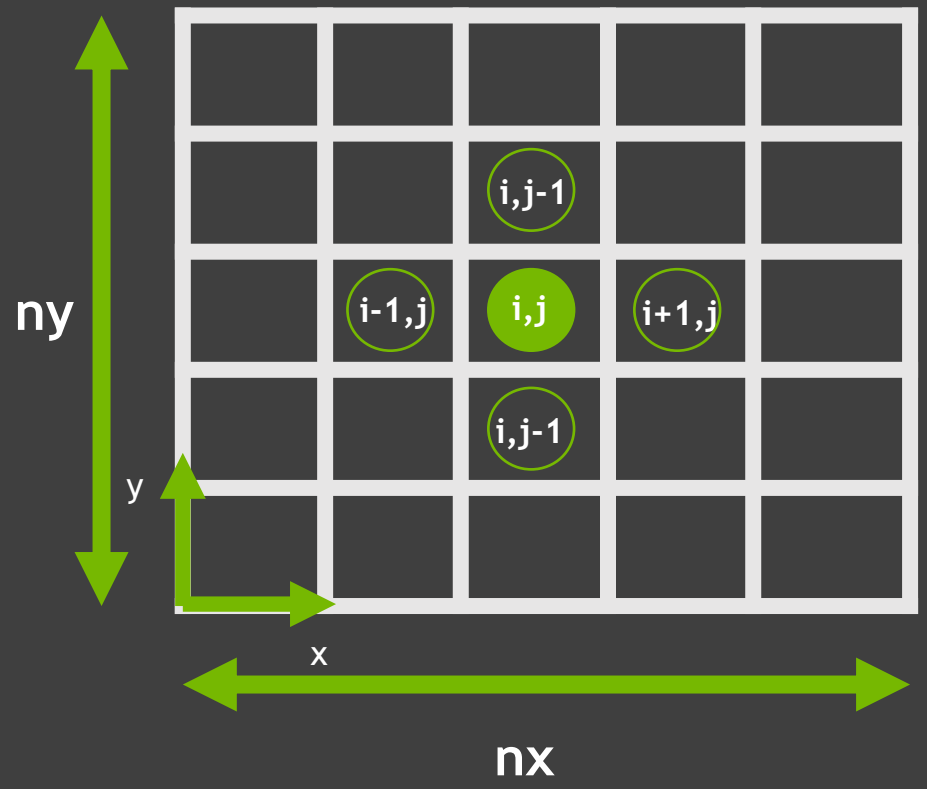
$$E(t) = \int_V \rho c T(x,t)\, dV$$



2D unsteady heat equation
Discretization



2D unsteady heat equation Stencil

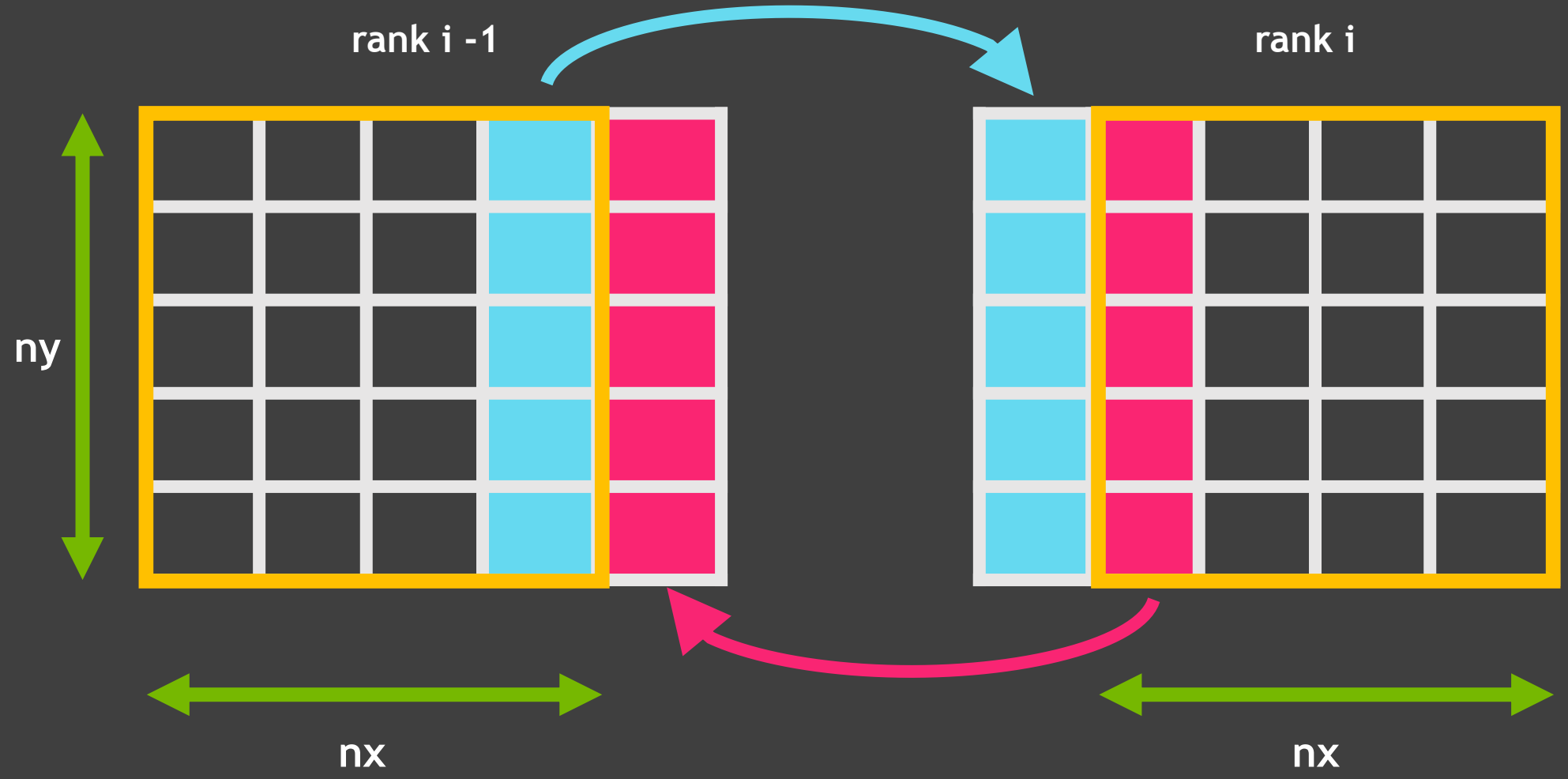


$$T(x_{i,j}, t + \Delta t) = \sum_{r=i-1}^{i+1} \sum_{s=j-1}^{j+1} c_{r,s}\, T(x_{r,s}, t)$$

Read Old + Write New

Read Old

2D unsteady heat equation
1D domain decomposition: halo exchange



2D unsteady heat equation Application structure

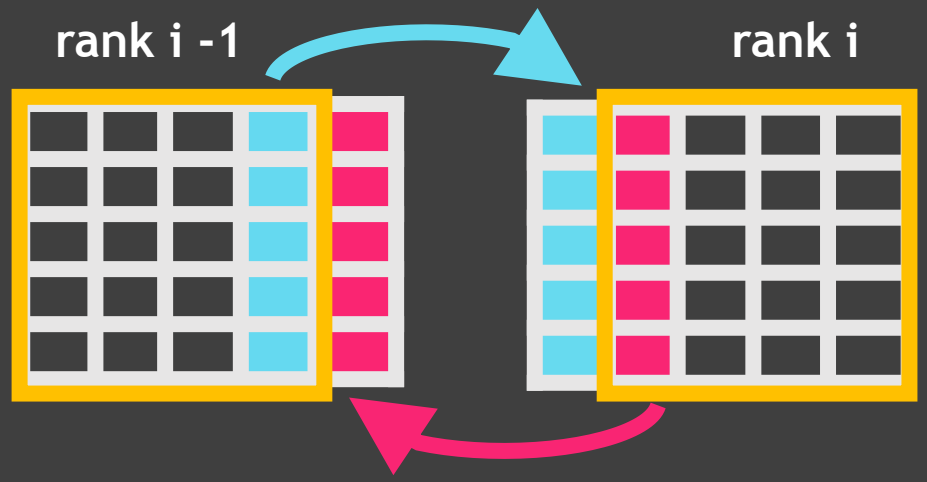
```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

2D unsteady heat equation Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

2D unsteady heat equation
Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```



2D unsteady heat equation
Application structure

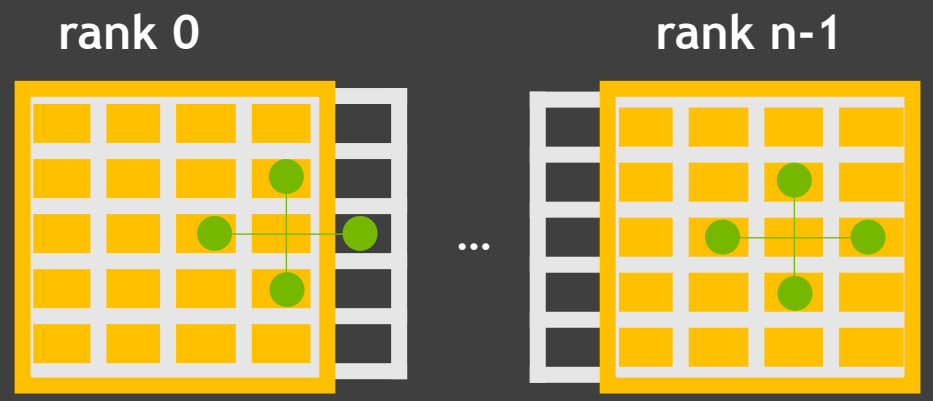
```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

2D unsteady heat equation
Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

2D unsteady heat equation
Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

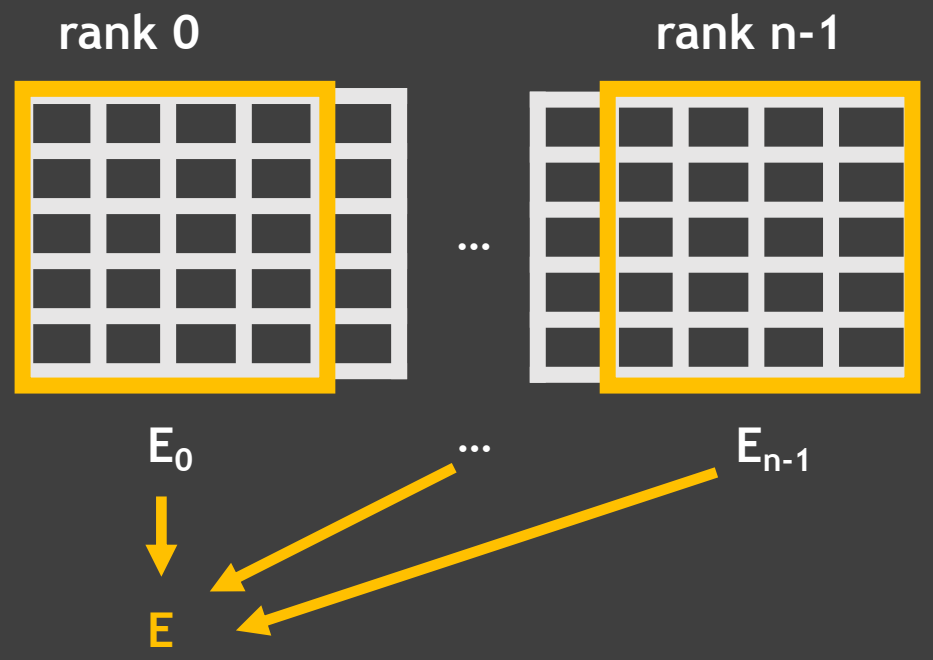


2D unsteady heat equation
Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```

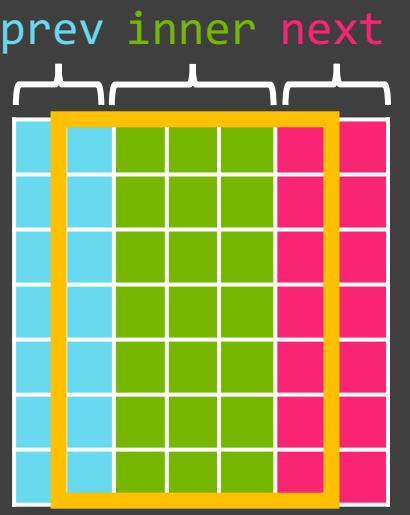
2D unsteady heat equation
Application structure

```
auto [nx, ny, niterations] = read_inputs(argc, argv);
auto [u, u_old] = alloc_data(nx, ny);
apply_initial_condition(u, u_old);

for (int it = 0; it < niterations; ++it) {
    exchange_halos();
    apply_boundary_conditions();
    float energy = apply_stencil();
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
write_file();
```



2D unsteady heat equation
Actual application structure

```
for (int it = 0; it < niterations; ++it) {
    float energy_prev = []{ exchange_halos(prev); apply_stencil(prev); }();
    float energy_next = []{ exchange_halos(next); apply_stencil(next); }();
    float energy_inner = apply_stencil(inner);
    float energy = energy_prev + energy_next + energy_inner;
    MPI_Reduce(&energy, …);
    if (rank == 0) print(energy);
}
```

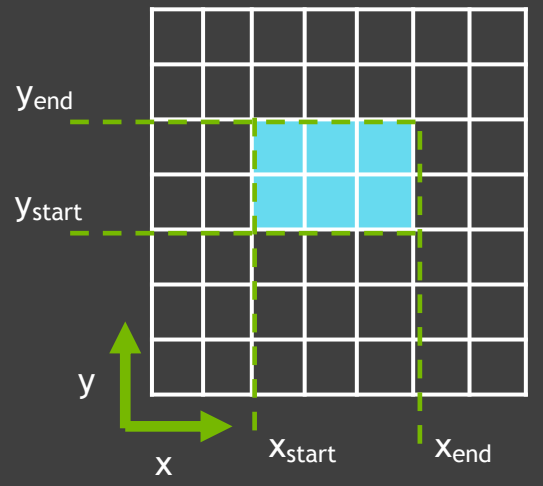


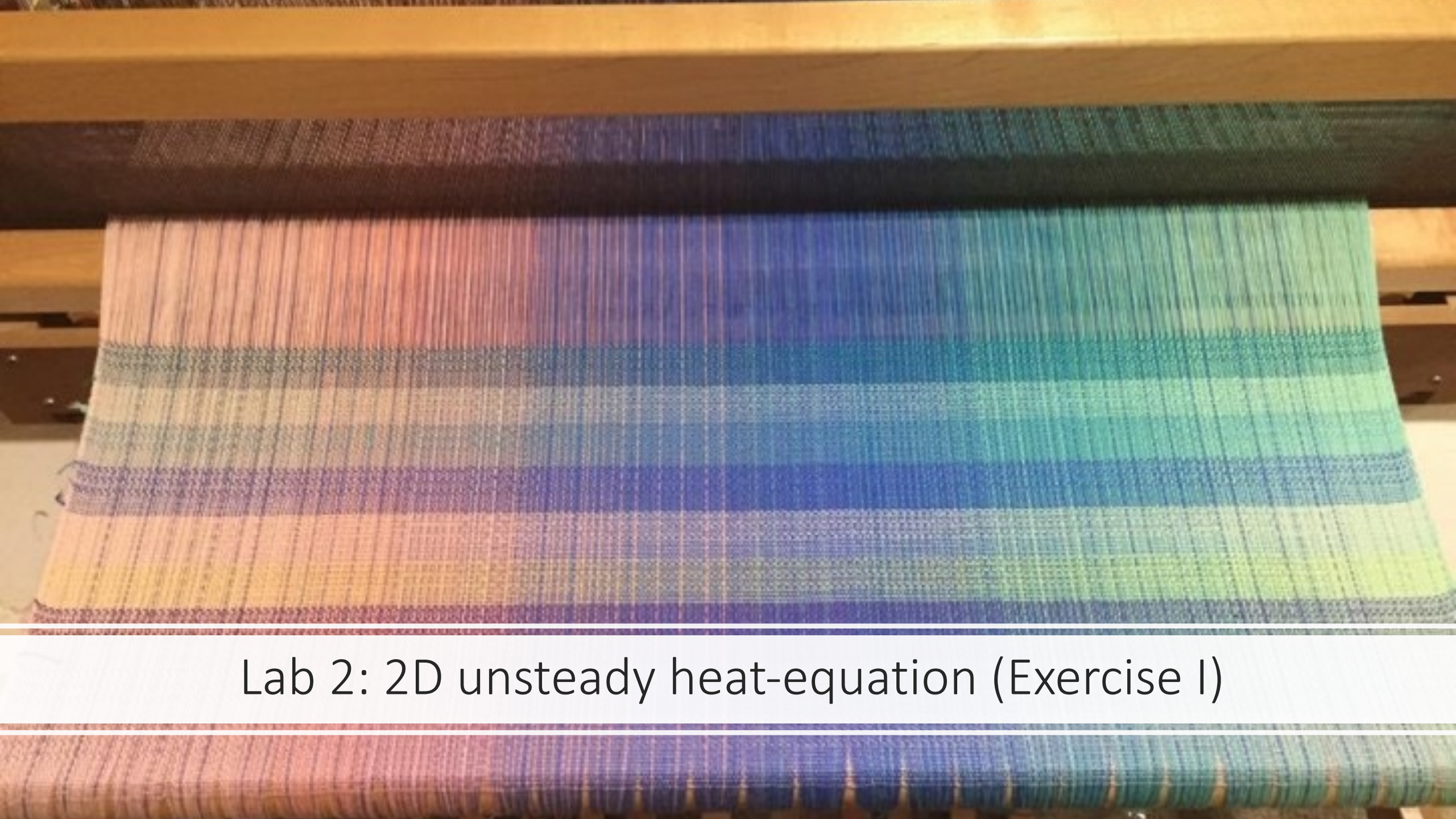
This structure will be useful in Exerise 2 to overlap computation with communication.

Boundary conditions handled inside apply_stencil…

2D unsteady heat equation
Grid kernel: updates elements of a grid tile

```
double apply_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 += stencil_one(u_new, u_old, x, y, p);
  return energy;
}

struct grid {
    // half-open ranges: [start, end)
    long x_start, x_end, y_start, y_end;
};
```

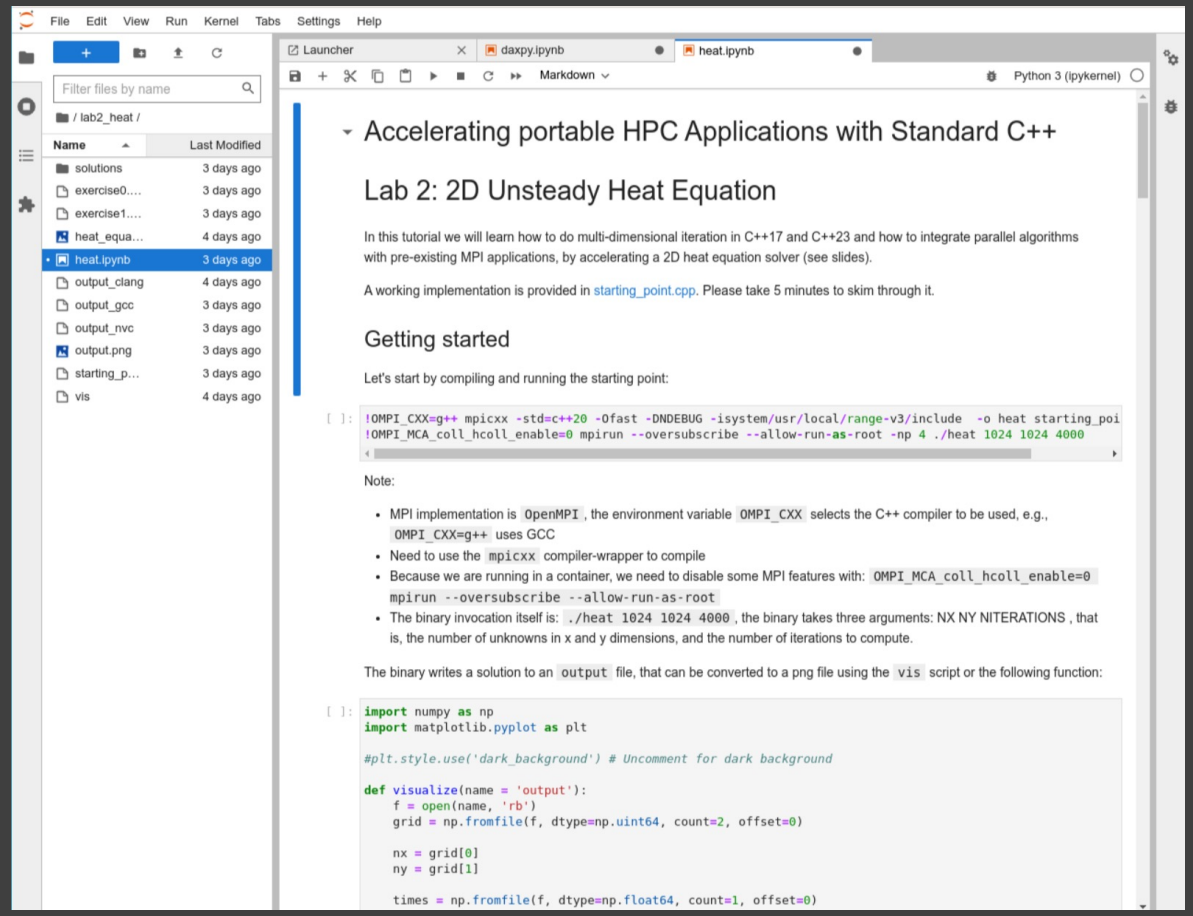




Lab 2: 2D unsteady heat-equation (Exercise I)

Lab 2: 2D unsteady heat equation Exercise I

- **Exercise 1:** parallelize the MPI implementation using the STL parallel algorithms.
  - This hybrid C++/MPI application parallelizes across all cores within one CPU socket, or all GPU resources with 1 MPI rank per GPU, depending on how it is compiled.



Lab 2: 2D unsteady heat equation
Exercise I: rewrite stencil to parallel algorithms

- **Before:**

```
double apply_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 += stencil_one(u_new, u_old, x, y, p);
  return energy;
}
```

Lab 2: 2D unsteady heat equation
Exercise I: rewrite stencil to parallel algorithms

- **After:**

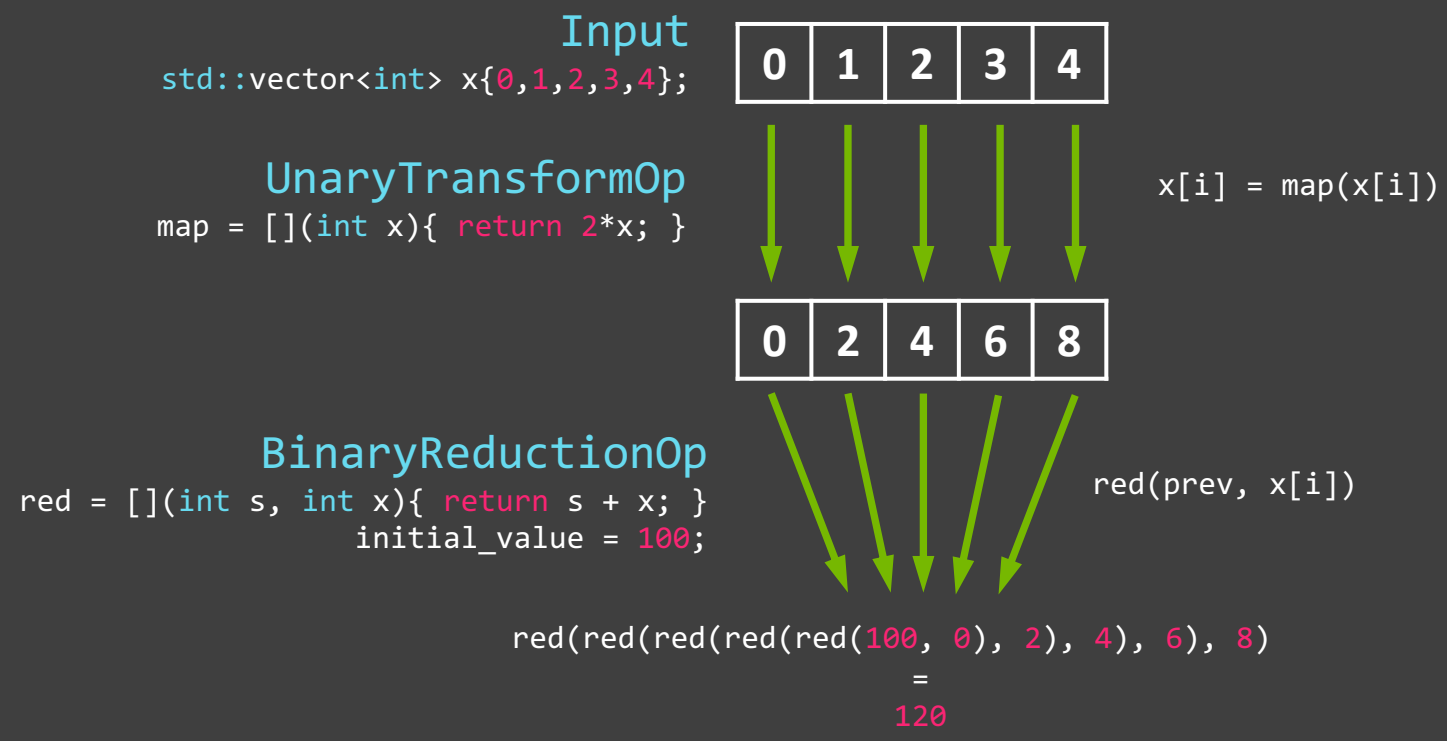
```
double apply_stencil(double* u_new, double* u_old, grid g, params p) {
  // TODO: use parallel algorithms
  … stencil_one(u_new, u_old, x, y, p) …
  return energy;
}
```

Three things to keep in mind:

1. Which algorithm to use? std::transform_reduce
2. Nested parallelism vs collapsing into a single algorithm
3. Multi-dimensional iteration with a collapsed algorithm

Lab 2: 2D unsteady heat equation std::transform_reduce

```
#include <numeric>
T std::transform_reduce(par, begin, end, initial_value, red, map);
```



Lab 2: 2D unsteady heat equation Stencil with std::transform_reduce?

The inner loop in our stencil application…

```
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);
return energy;
```

…can be written using std::transform_reduce as follows…

…but is only parallel over ys!

```
double energy = 0.;
for (long x = g.x_start; x < g.x_end; ++x) {
  auto ys = std::views::iota(g.y_start, g.y_end);
  energy += std::transform_reduce(
      std::execution::par, ys.begin(), ys.end(),
      std::plus<>{},
      [=](long y) { return stencil(u_new, u_old, x, y, p); }
  );
}
return energy;
```

Lab 2: 2D unsteady heat equation Nested parallelism

Just call std::transform_reduce twice then?

```
auto xs = std::views::iota(g.x_start, g.x_end);
double energy = std::transform_reduce(
    std::execution::par,
    xs.begin(), xs.end(),
    std::plus<>{},
    [=](long x) {
      auto ys = std::views::iota(g.y_start, g.y_end);
      return std::transform_reduce(
        std::execution::par, ys.begin(), ys.end(),
        std::plus<>{},
        [=](long y) {
            return stencil(u_new, u_old, x, y, p);
      });
});
return energy;
```

Lab 2: 2D unsteady heat equation
Nested parallelism is expensive

Just call std::transform_reduce twice then?

Launching parallel work from within parallel work is expensive and often run sequentially!

```
auto xs = std::views::iota(g.x_start, g.x_end);
double energy = std::transform_reduce(
    std::execution::par,
    xs.begin(), xs.end(),
    std::plus<>{},
    [=](long x) {
      auto ys = std::views::iota(g.y_start, g.y_end);
      return std::transform_reduce(
        std::execution::par, ys.begin(), ys.end(),
        std::plus<>{},
        [=](long y) {
            return stencil(u_new, u_old, x, y, p);
      });
});
return energy;
```

Lab 2: 2D unsteady heat equation Collapsed algorithm

What we want is a single parallel algorithm that processes all elements…

```
auto indices = ???;
double energy = std::transform_reduce(
    std::execution::par,
    indices.begin(), indices.end(),
    std::plus<>{},
    [=](auto indices) {
      auto [x, y] = indices;
      return stencil(u_new, u_old, x, y, p);
});
return energy;
```

C++23 std::views::cartesian_product
Produces a range of tuples

```
auto xs = std::views::iota(0, N);
auto ys = std::views::iota(0, M);
auto indices = std::views::cartesian_product(xs, ys);
```

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

Lab 2: 2D unsteady heat equation Collapsed algorithm with cartesian_product

What we want is a single parallel algorithm that processes all elements…

…and we achieve this by mapping multi-dimensional iteration to one-dimensional iteration using std::views::cartesian_product!

```
auto xs = std::views::iota(g.x_start, g.x_end);
auto ys = std::views::iota(g.y_start, g.y_end);
auto indices = std::views::cartesian_product(xs, ys);

double energy = std::transform_reduce(
    std::execution::par,
    indices.begin(), indices.end(),
    std::plus<>{},
    [=](auto indices) {
      auto [x, y] = indices;
      return stencil(u_new, u_old, x, y, p);
});
return energy;
```

Lab 2: 2D unsteady heat equation
But what about C++17 and C++20 ?

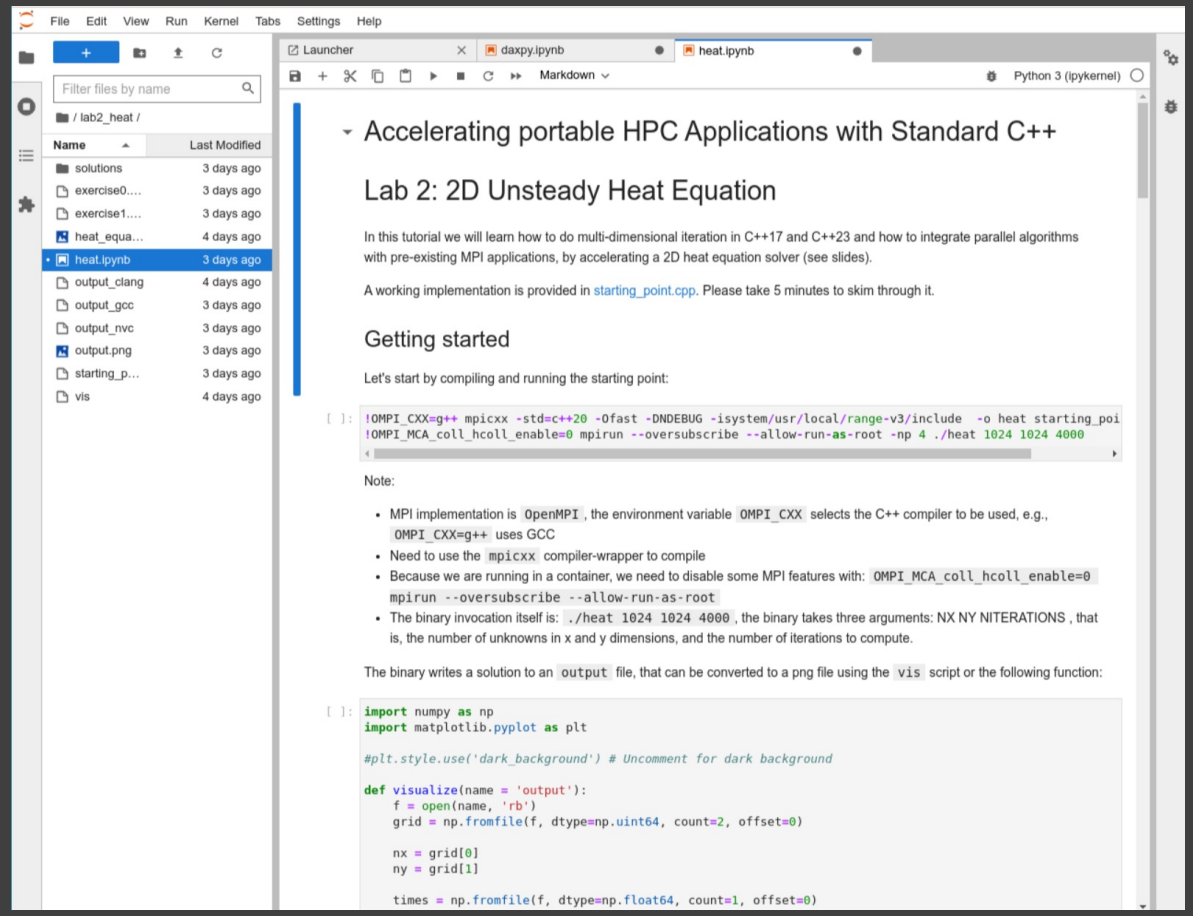
Use a range library that provides cartesian_product!

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

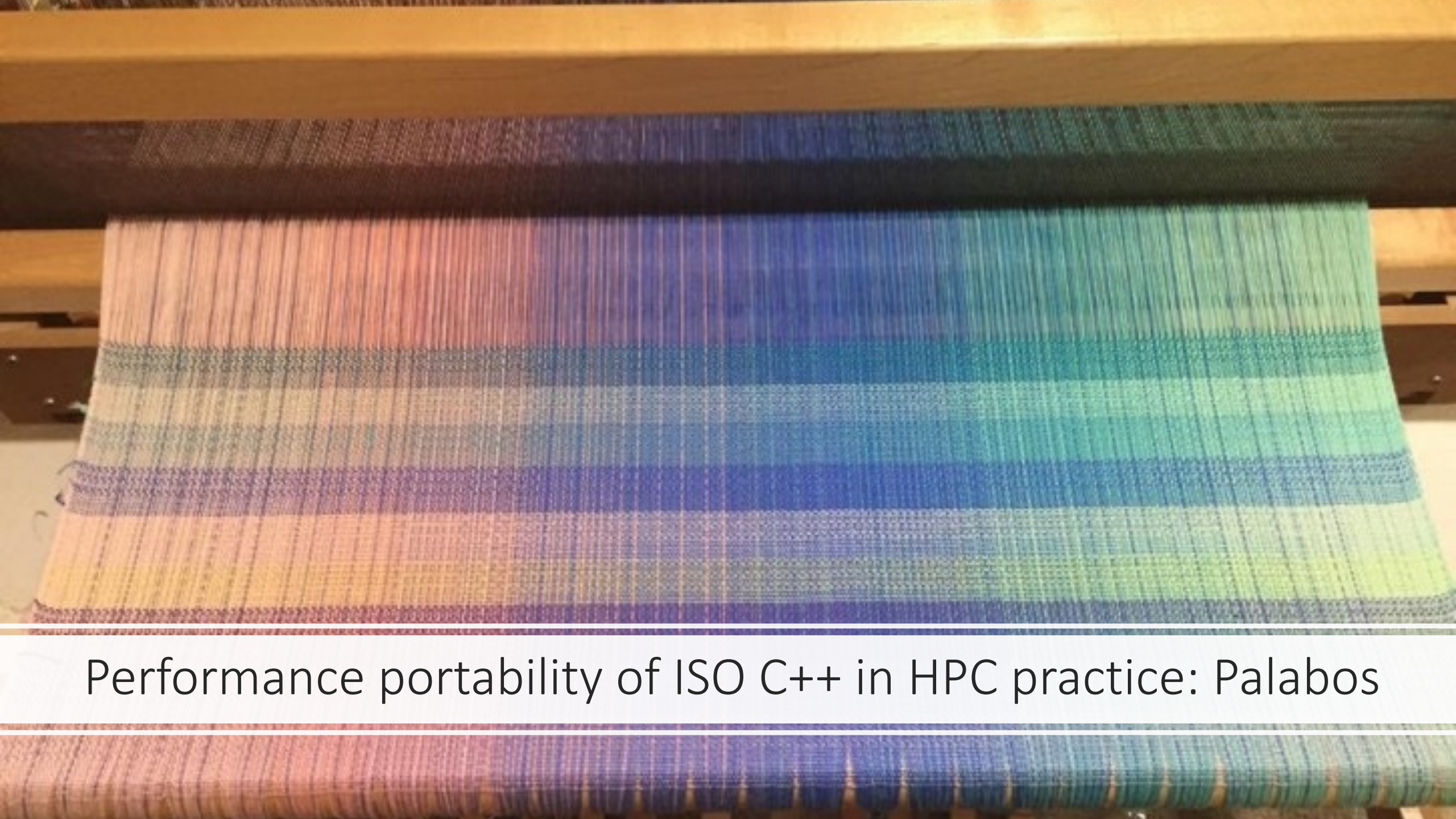
```
auto xs = std::views::iota(g.x_start, g.x_end);
auto ys = std::views::iota(g.y_start, g.y_end);
auto indices = std::views::cartesian_product(xs, ys);

double energy = std::transform_reduce(
    std::execution::par,
    indices.begin(), indices.end(),
    std::plus<>{},
    [=](auto indices) {
      auto [x, y] = indices;
      return stencil(u_new, u_old, x, y, p);
});
return energy;
```

Lab 2: 2D unsteady heat equation Exercise I

- **Exercise 1:** parallelize the MPI implementation using the STL parallel algorithms.
  - This hybrid C++/MPI application parallelizes across all cores within one CPU socket, or all GPU resources with 1 MPI rank per GPU, depending on how it is compiled.



Lab 2: 2D heat equation (Part I)
Solutions (DEMO)



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

Palabos: Multi-physics simulation framework

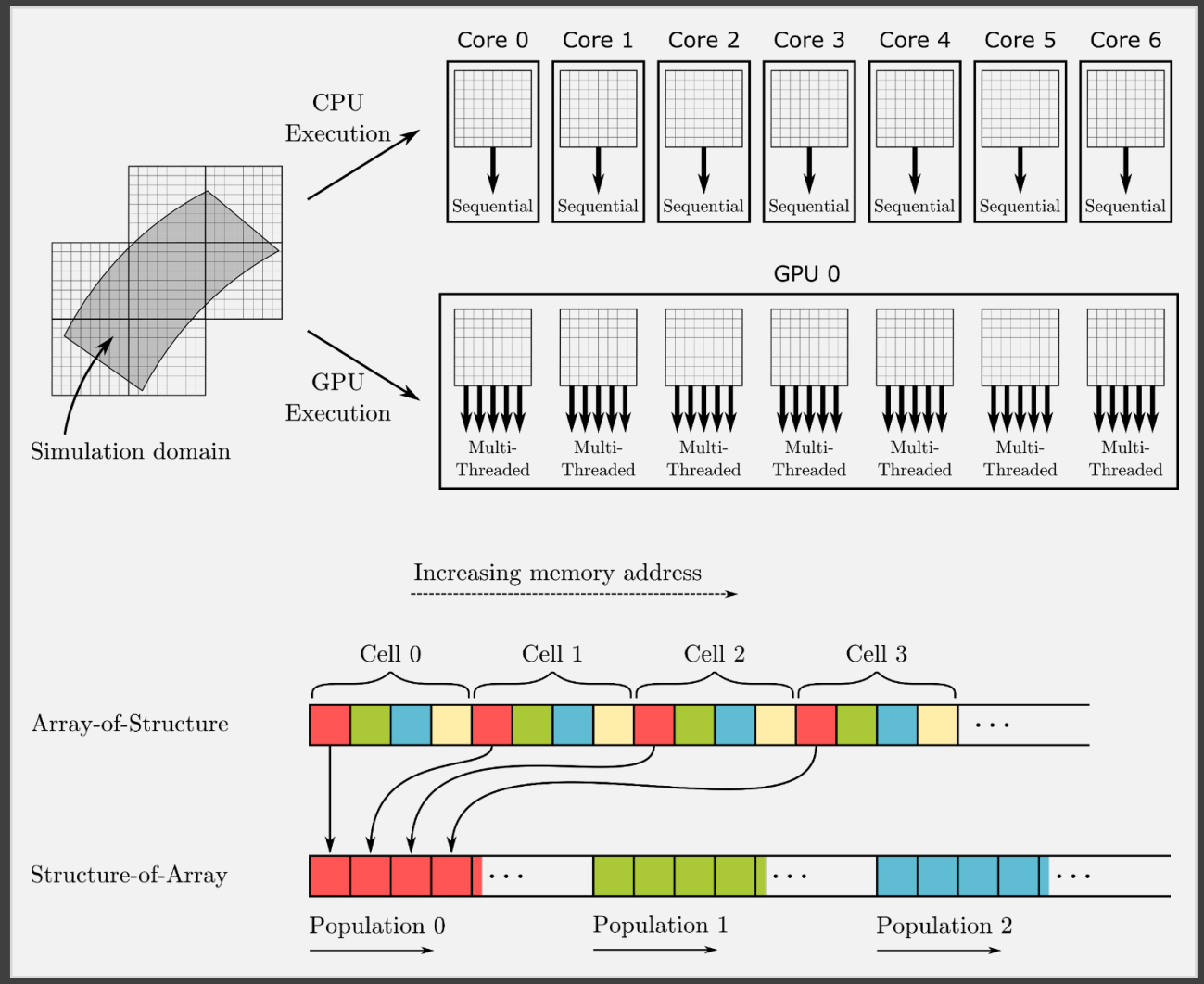
- Developed at University of Geneva
- Based on Lattice Boltzmann methods
- Multi-phase flows, flow through porous media, etc.

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

A typical algorithm in Palabos

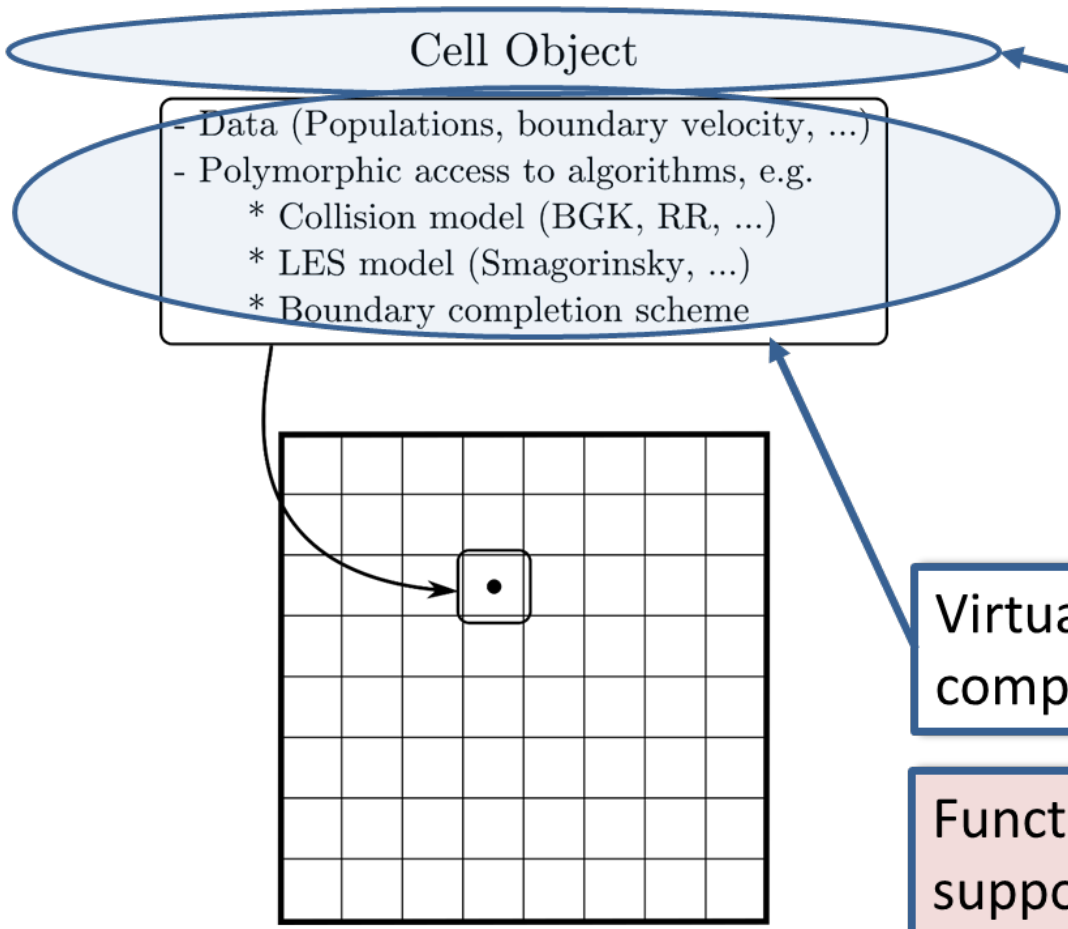
```
std::for_each(par_unseq, begin(c), end(c),
[c = c.data()](double& i) {
  int i = &x - v_ptr;

  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 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”.

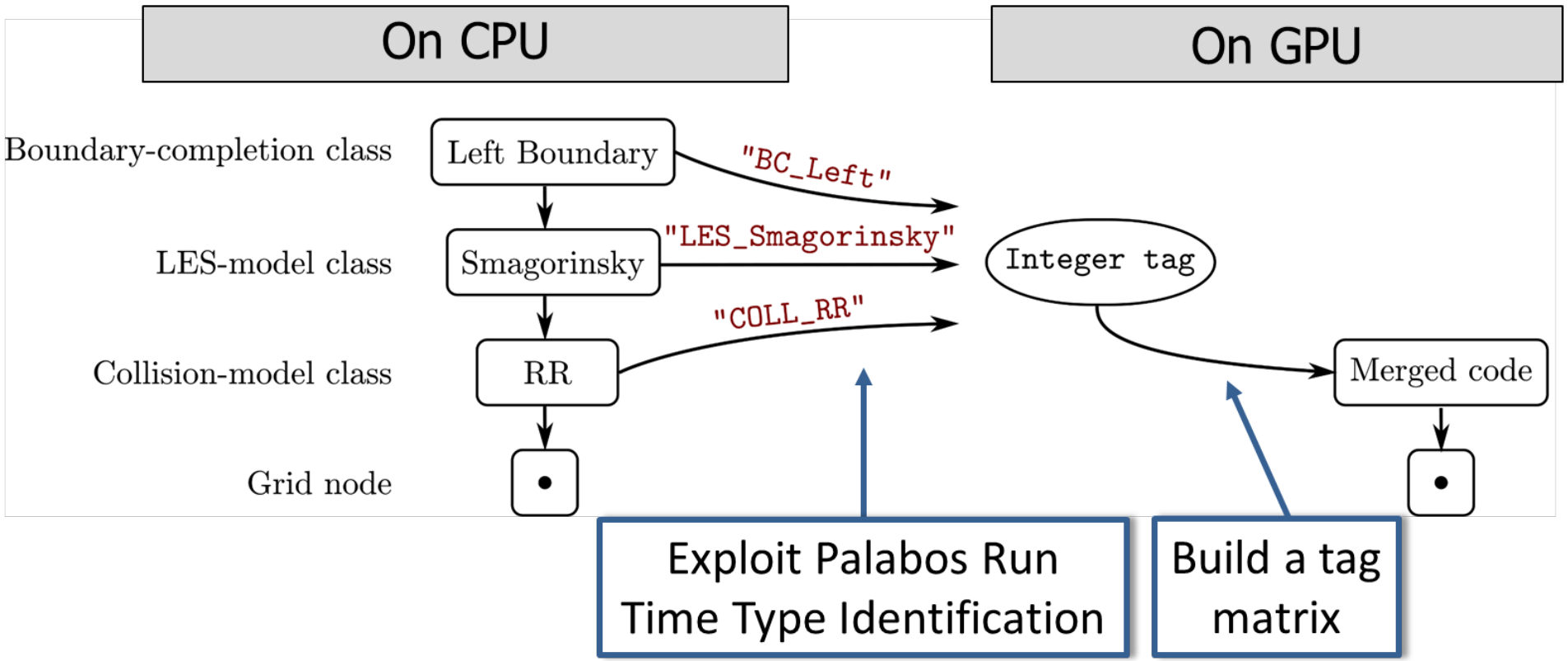
GPUs don’t like the resulting memory layout.

Virtual function calls to different model components.

Function-pointer call mechanism is not supported.

Polymorphic objects → Tag matrix

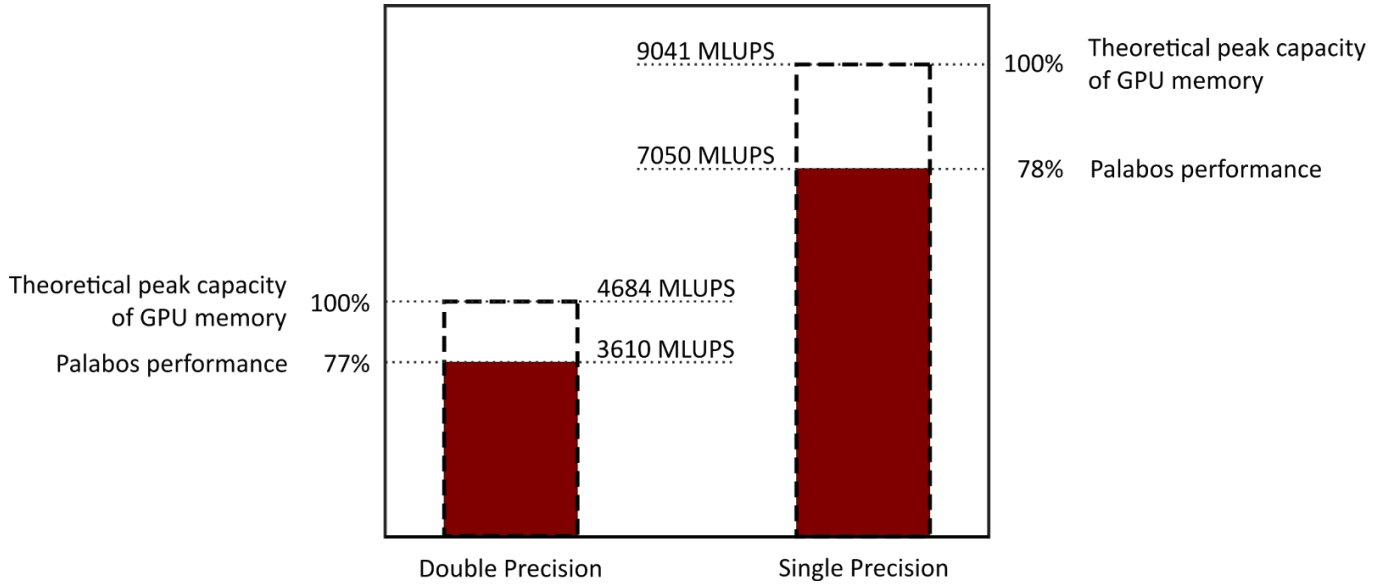
The tag matrix suits the GPU better and can be generated automatically



Palabos
1GPU Performance

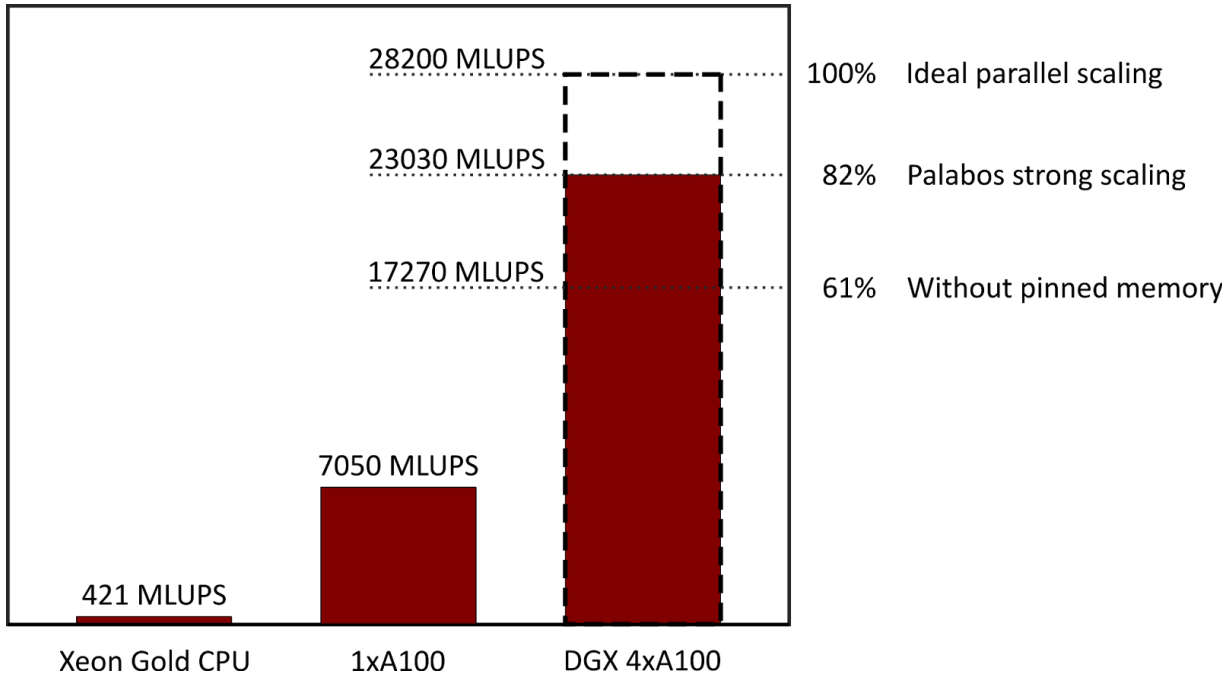
Performance model:

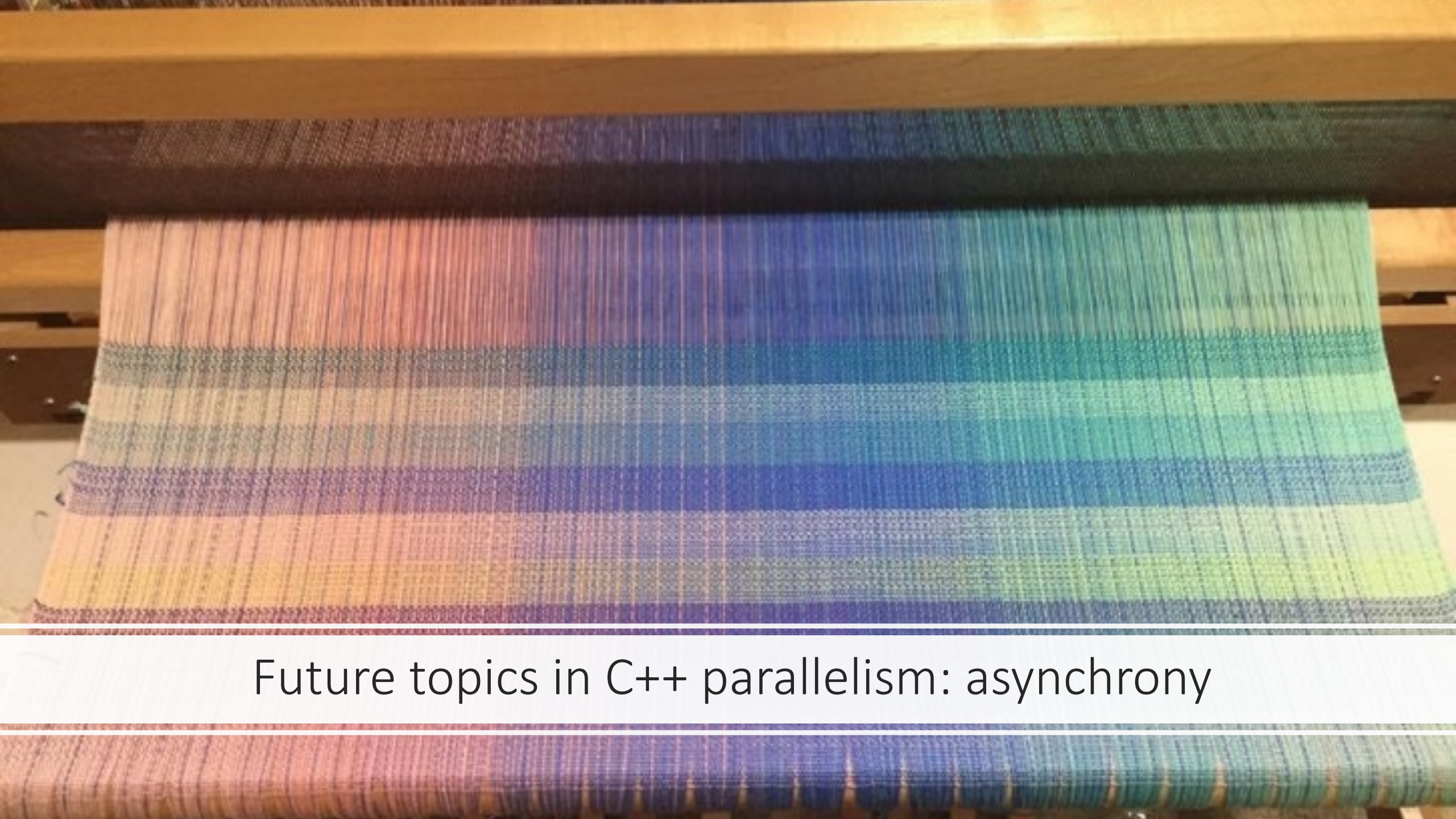
- Lattice Boltzmann D2Q19:
  19 fields + 1 index
- Memory bound
- Mega Lattice Updates / s:
  - BW_DRAM / (2 * 20 * Float Bytes)
  - A100-40: ~1500 GB/s / (2 * 20 * 8 B) ~= 4684 MLUPs
- State of the art: ~80% SOL



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.

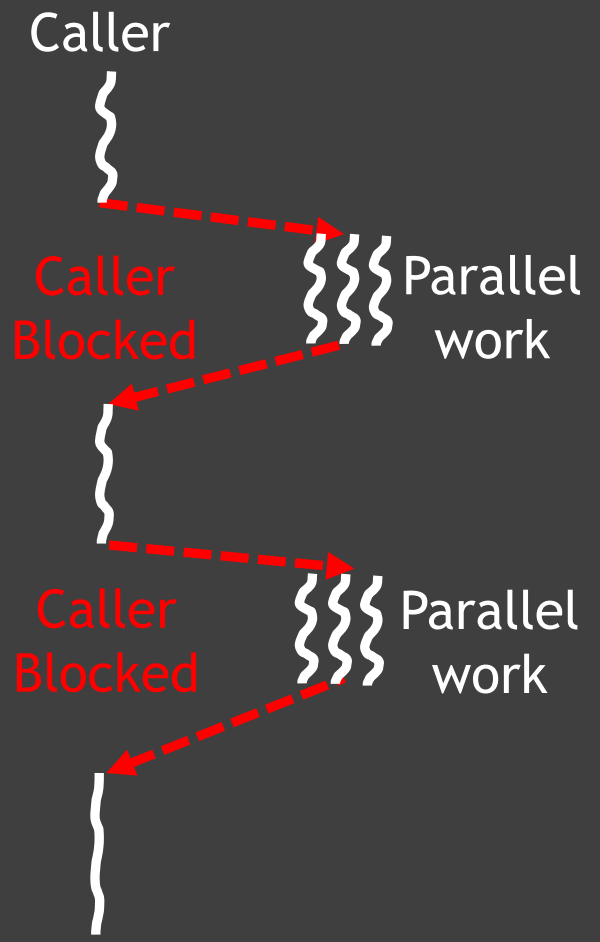




Future topics in C++ parallelism: asynchrony

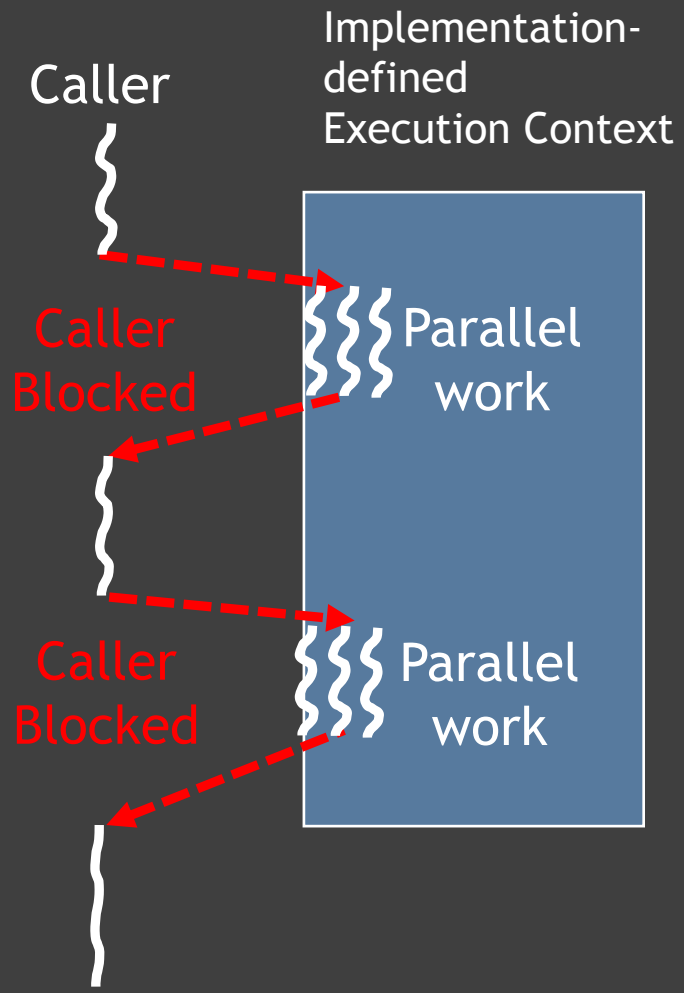
Limitations of parallel algorithms
Fork-join synchronous

```
std::vector<int> x {…};

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

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

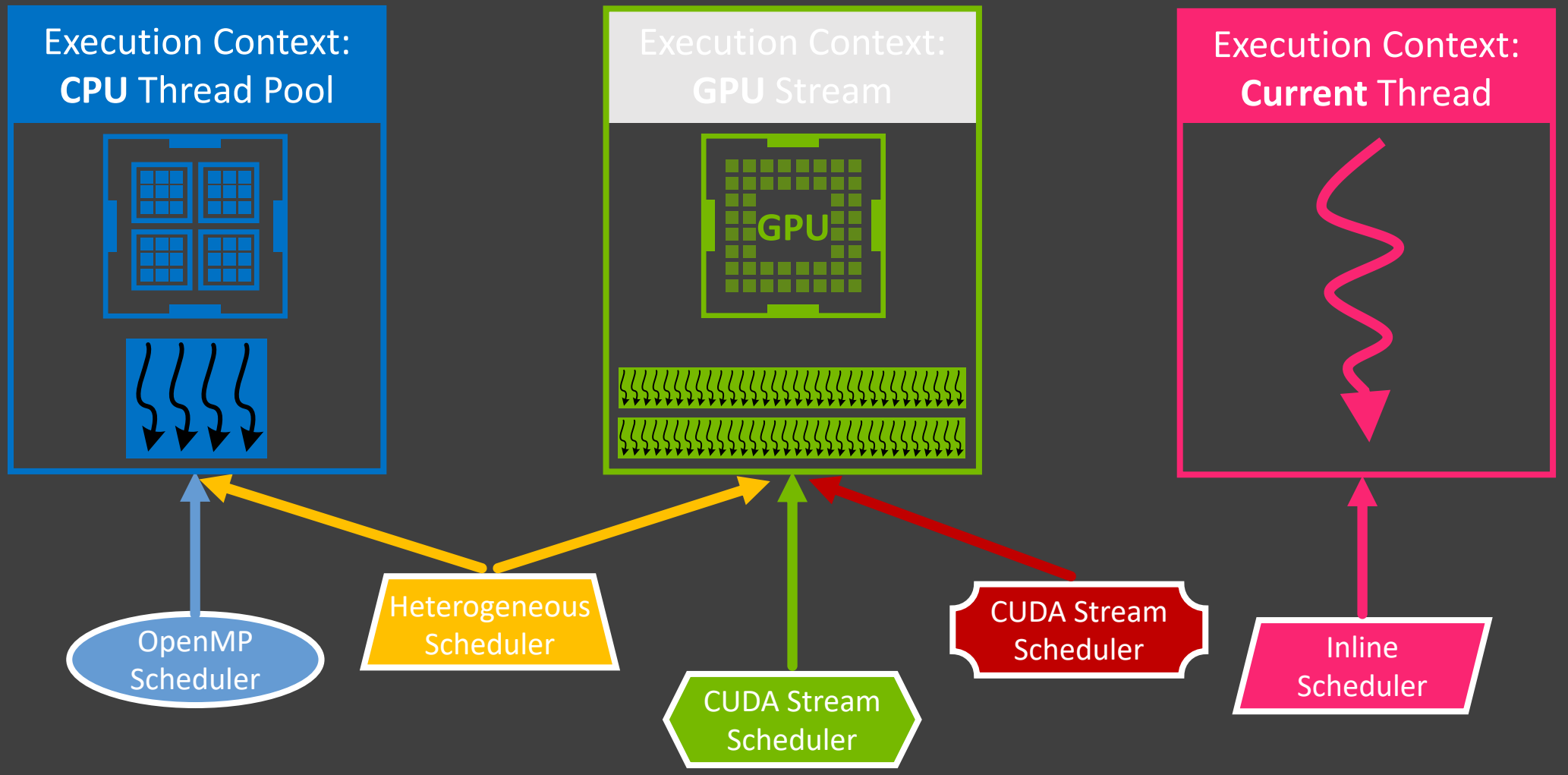


Limitations of parallel algorithms
Fork-join synchronous

```
std::vector<int> x {…};

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


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

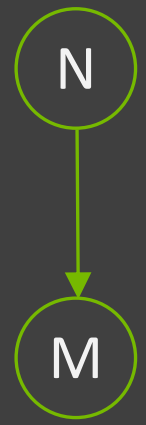


P2300R4: std::execution



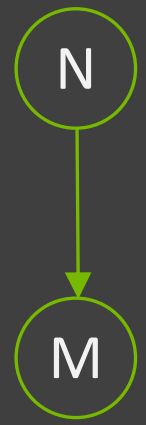
P2300R4: std::execution

```
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);
```



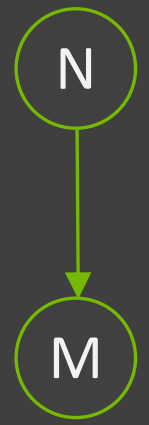
P2300R4: std::execution

```
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);
```



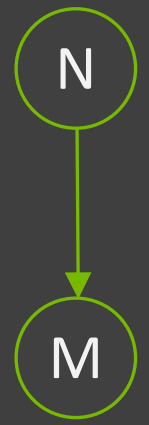
P2300R4: std::execution

```
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);
```



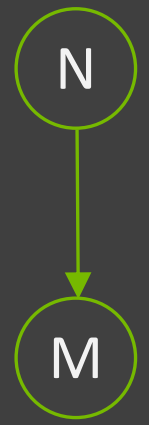
P2300R4: std::execution

```
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);
```



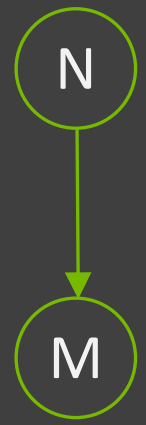
P2300R4: std::execution

```
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);
```



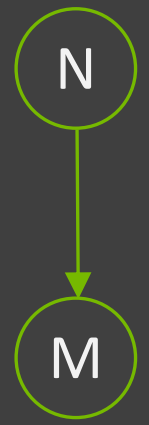
P2300R4: std::execution

```
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);
```



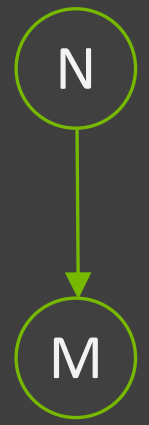
P2300R4: std::execution

```
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);
```

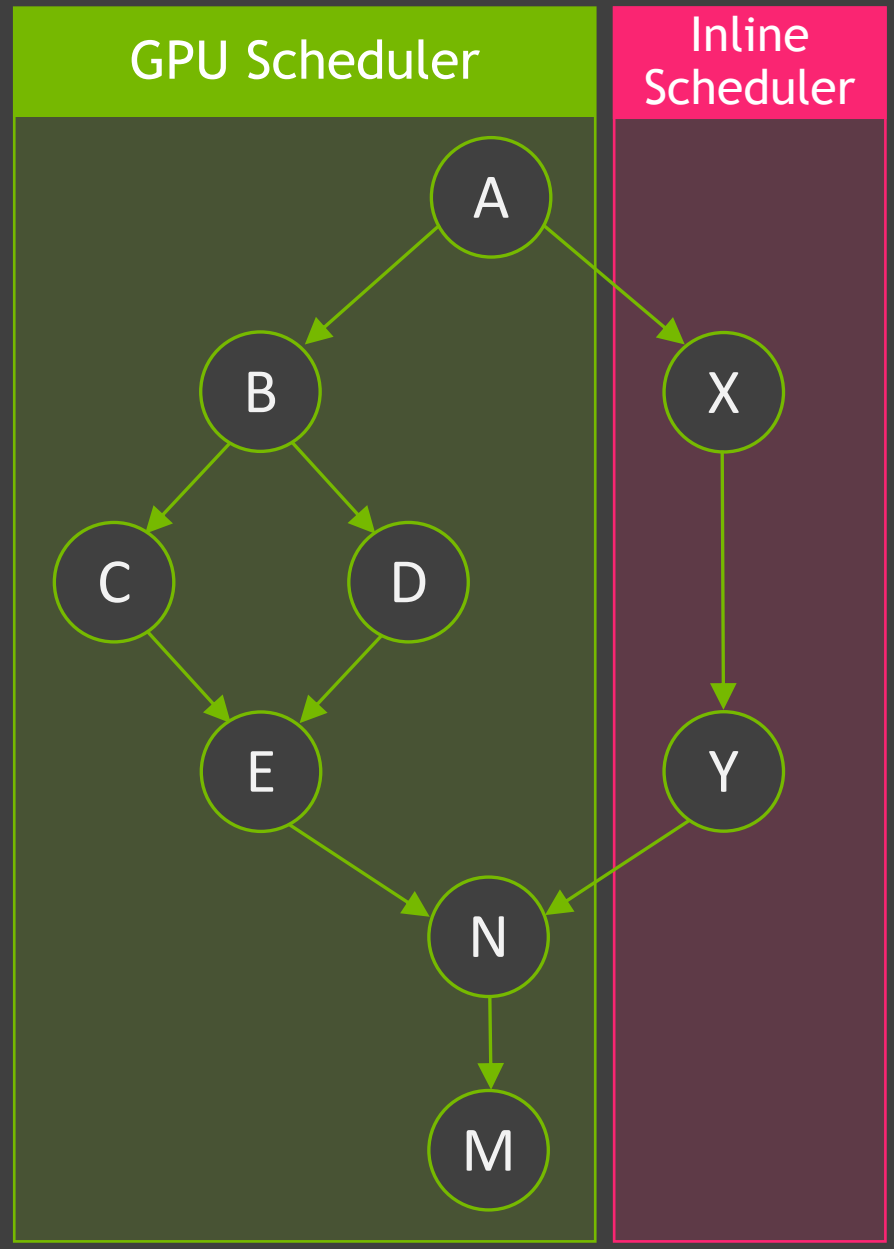


std::execution: P2300R4: std::execution Structured concurrency and parallelism

```
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);
```

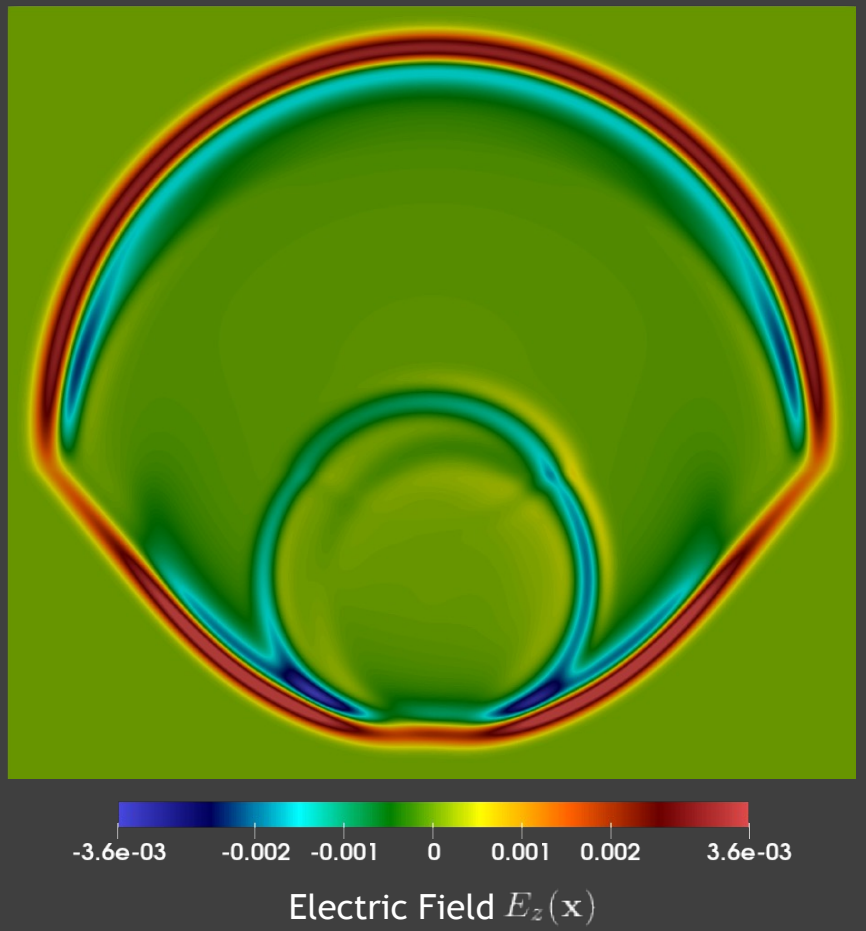
P2300R4: std::execution

```
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);
```



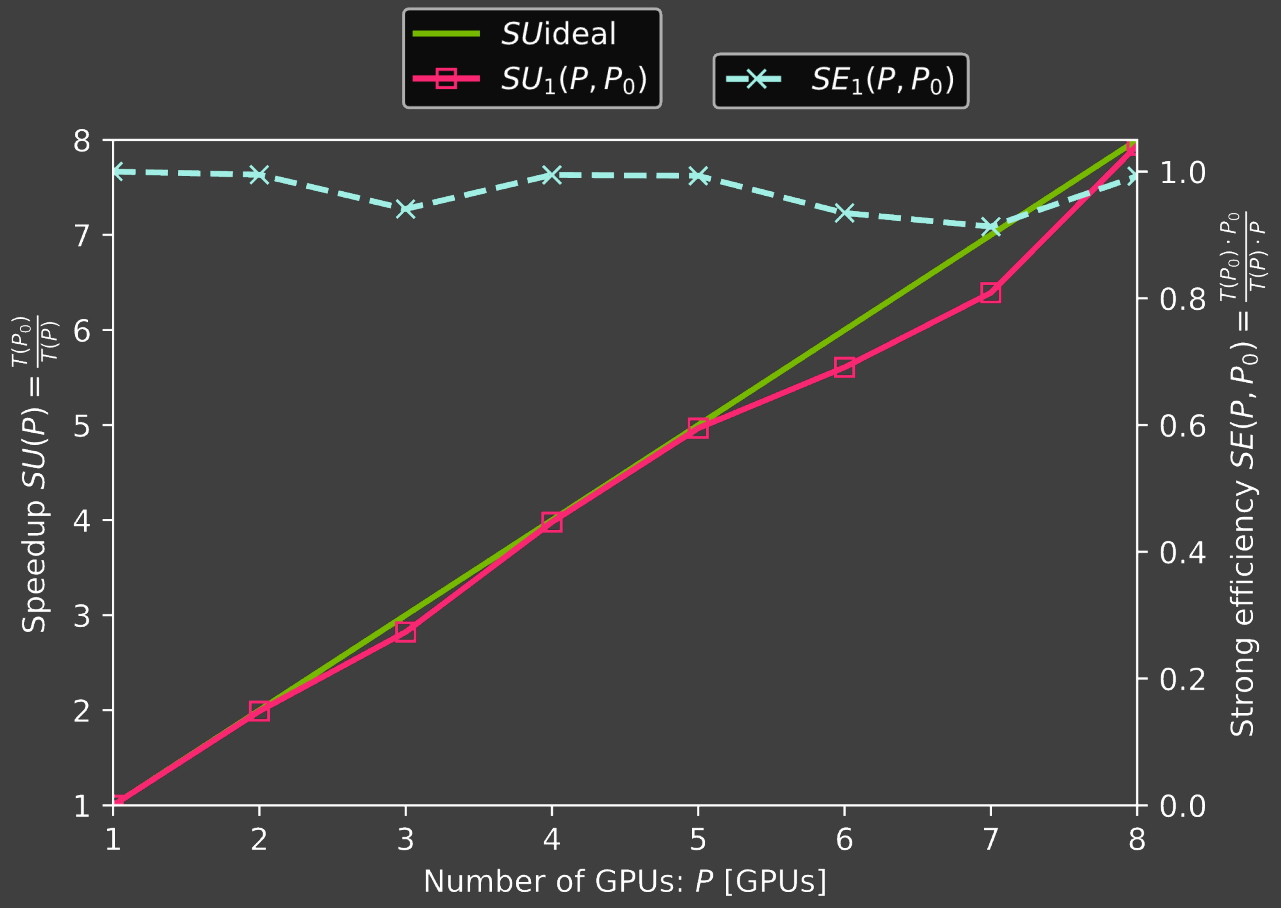
Maxwell Equations (DEMO)
Compute graphs

```
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);
```



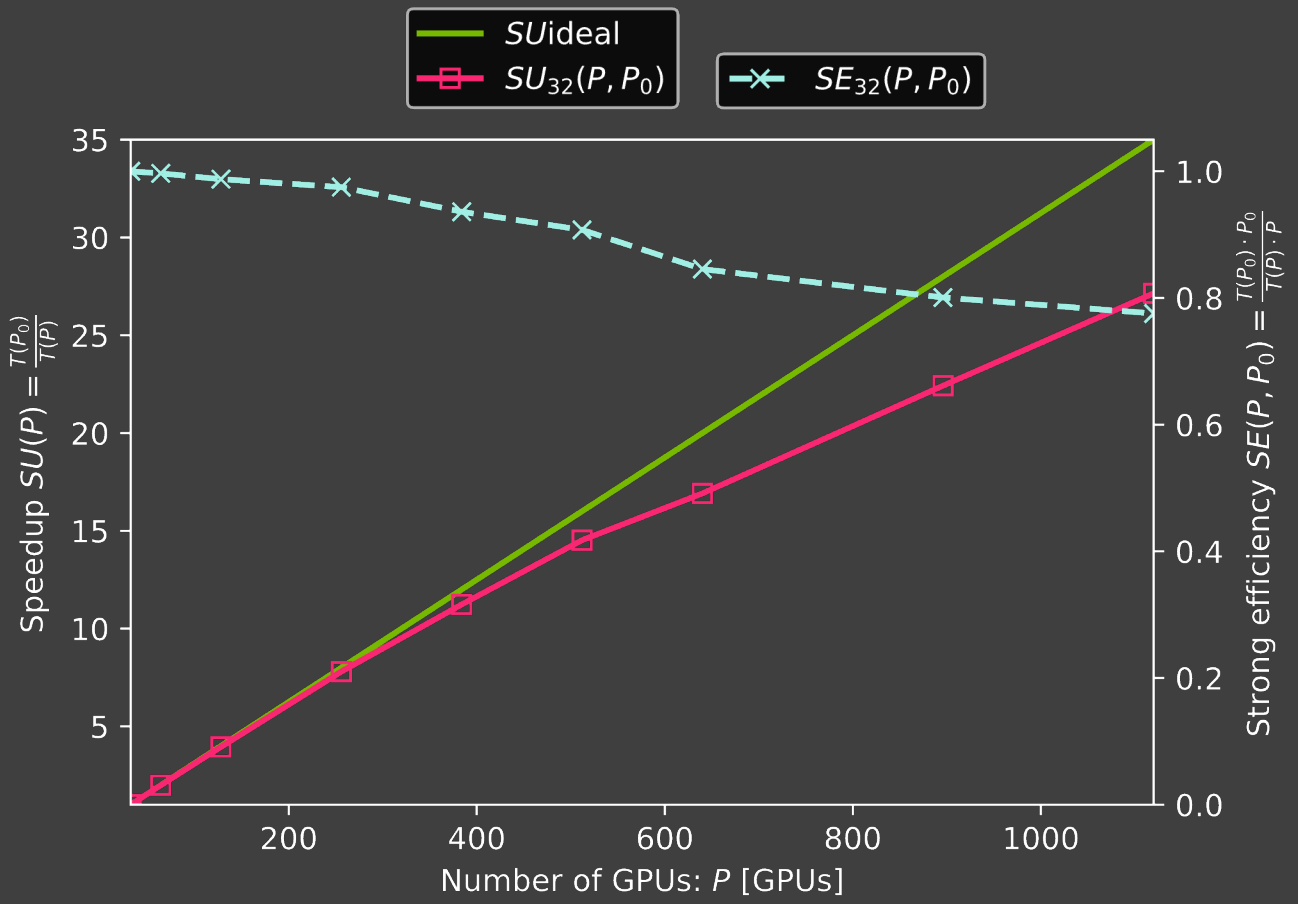
Maxwell Equations (DEMO)
DGX-A100 640 Strong scaling

- 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



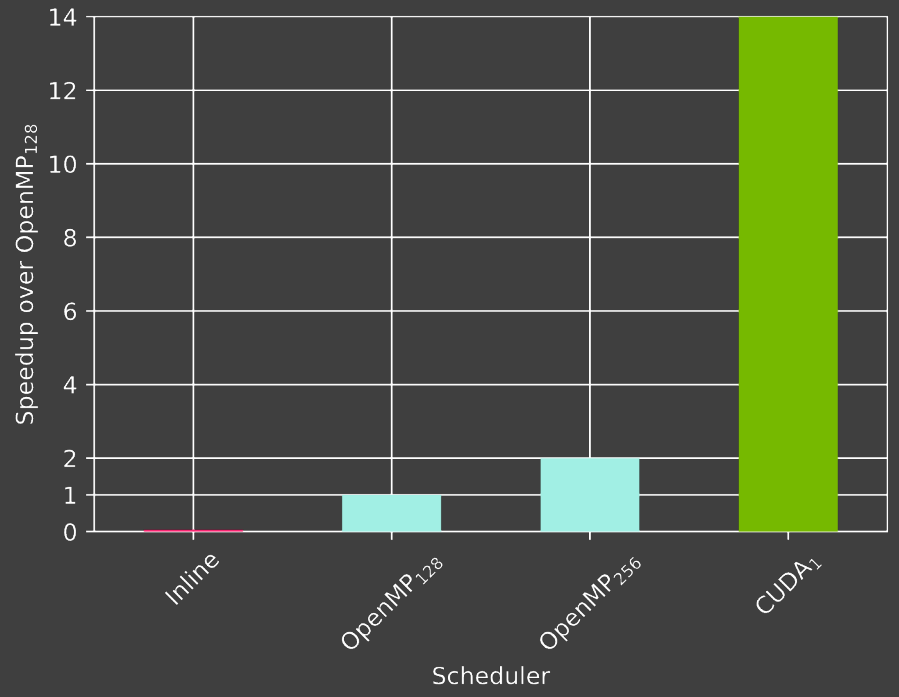
Maxwell Equations (DEMO)
Selene SuperPOD Strong scaling

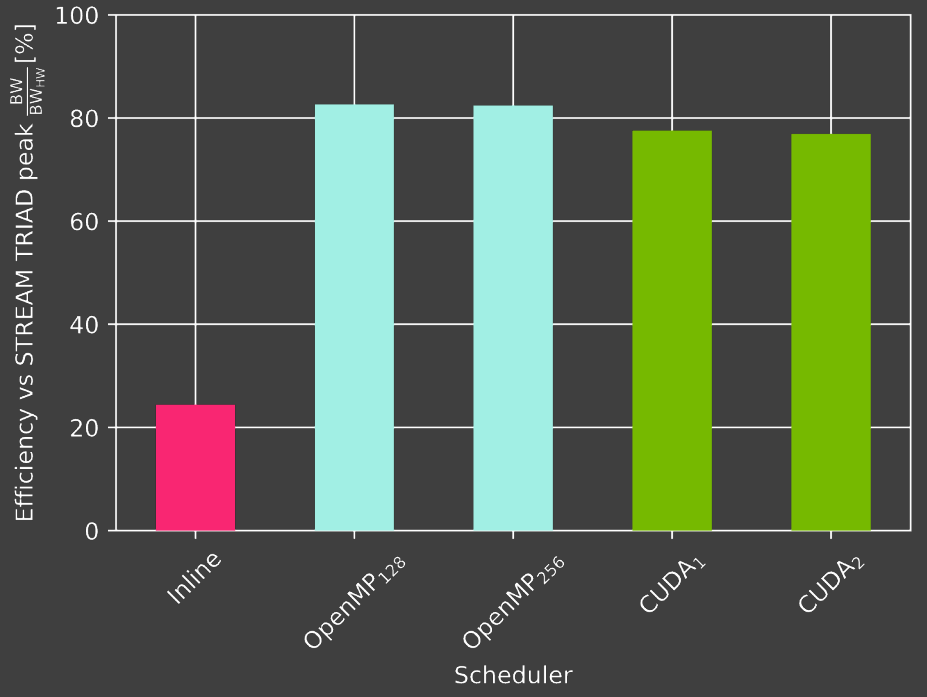
- #6 Top500 | #10 Green500 | #1 MLPerf | #5 HPCG
- 4x NVIDIA SuperPODs full fat-tree connected
- 560x NVIDIA DGX-A100 640 (4x 140)
- 4480x NVIDIA A100-SXM4-80 GPUs



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));
```

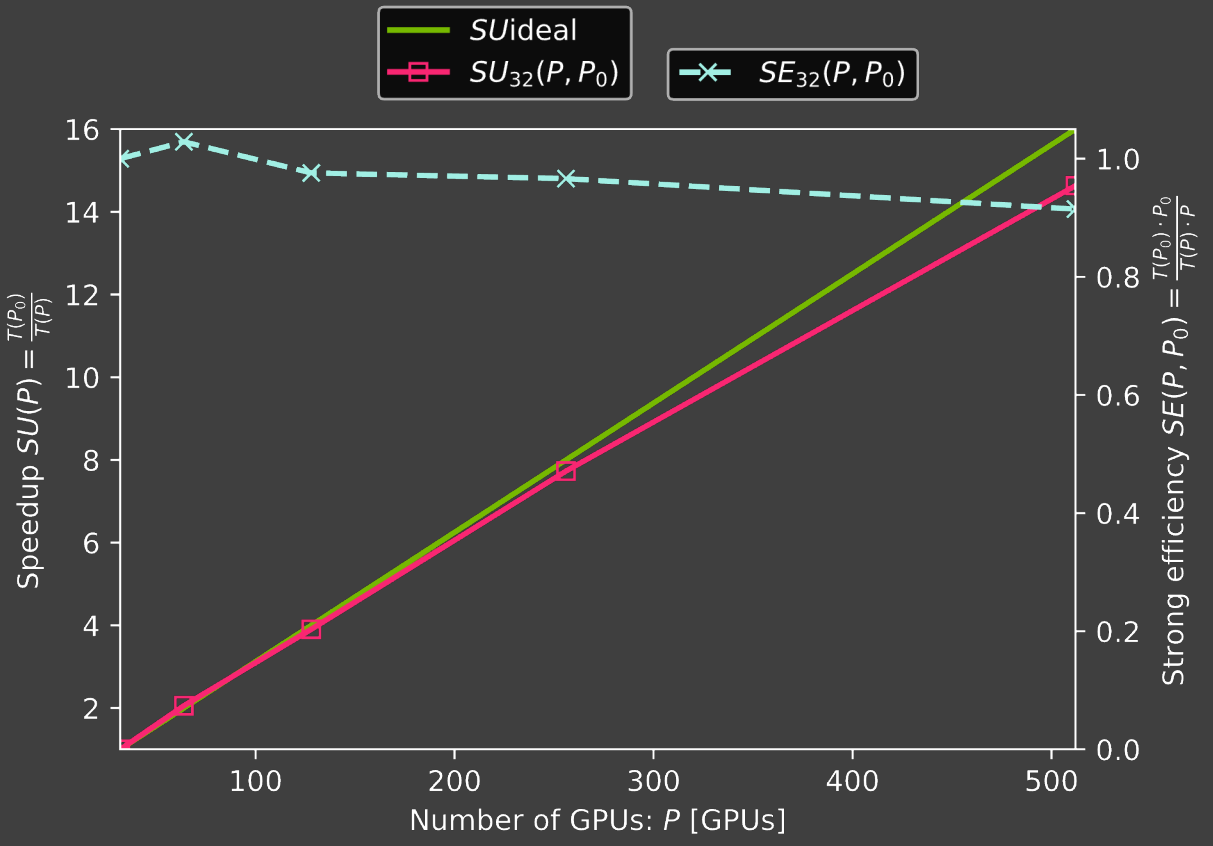


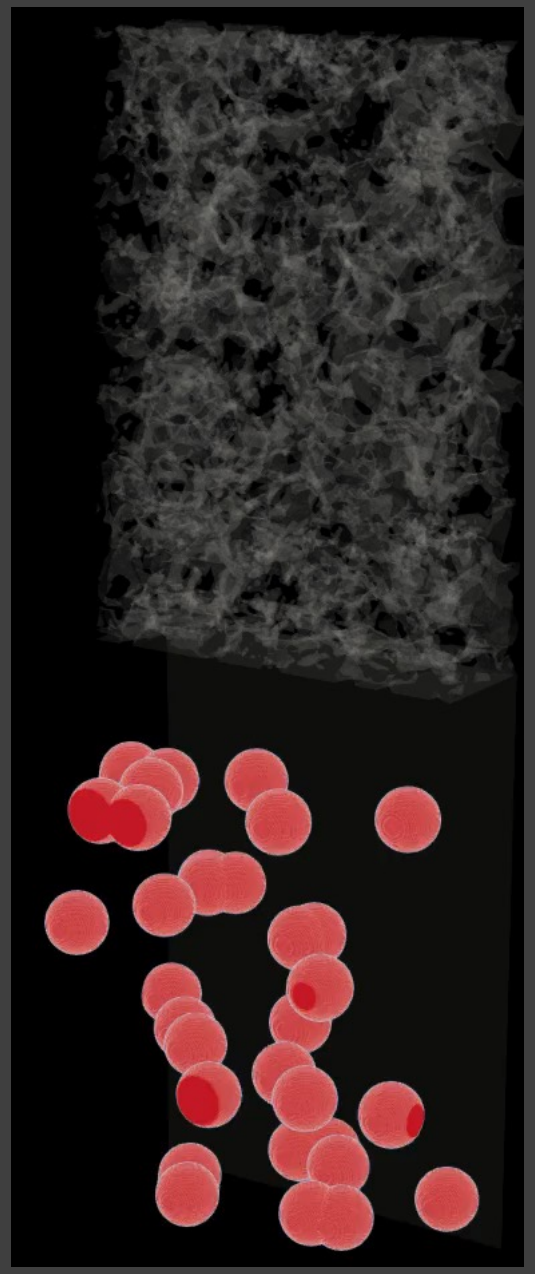


- CPUs: AMD EPYC 7742 CPUs, GPUs: NVIDIA A100-SXM4-80
- Inline (1 CPU HW thread), OpenMP-128 (1x CPU), OpenMP-256 (2x CPUs), Graph (1x GPU), Multi-GPU (2x GPUs)
- clang-12 with -O3 -DNDEBUG -mtune=native -fopenmp

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

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: linear algebra & multi-dimensional arrays

Proposals

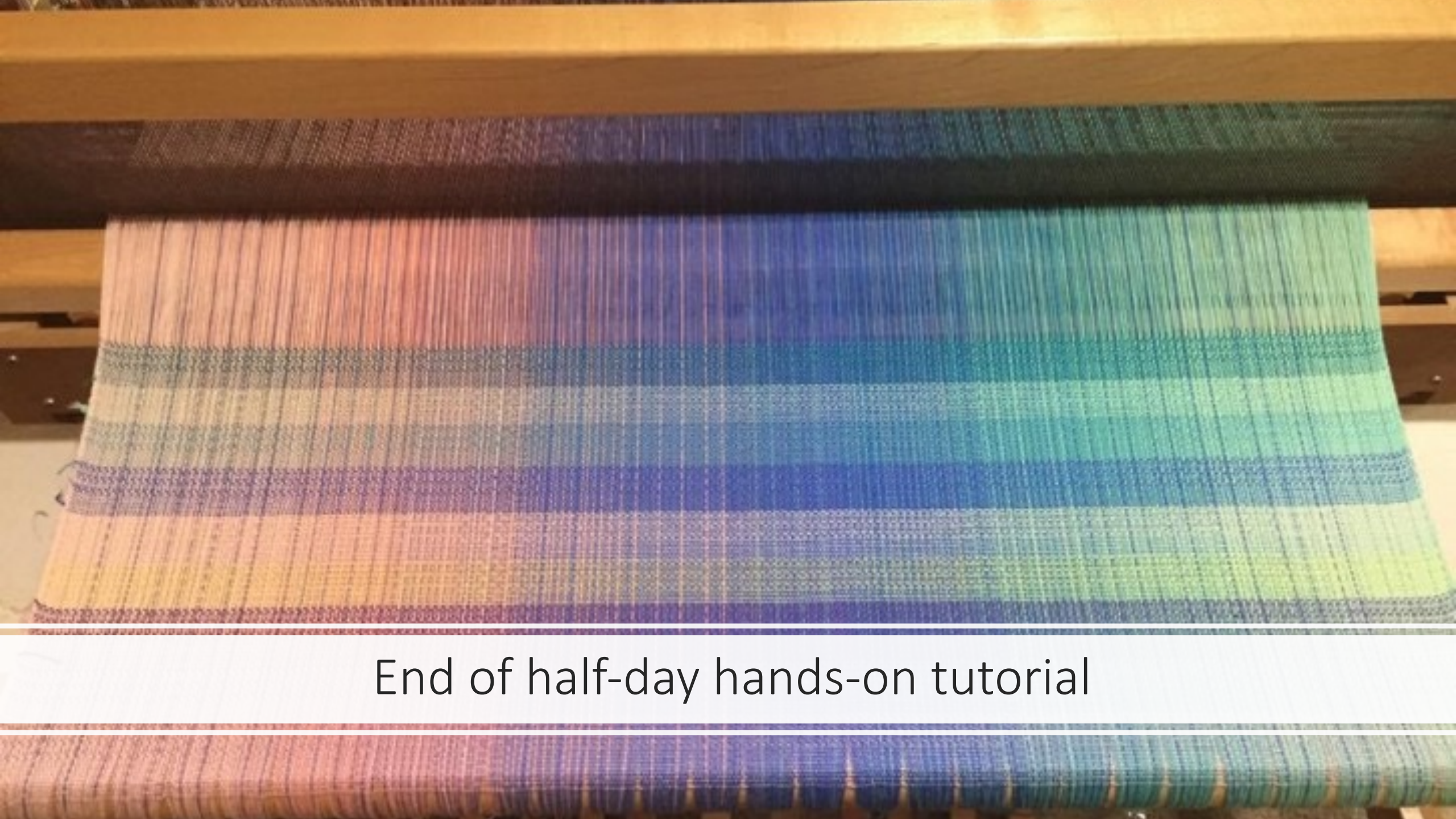
- P1637 A free function linear algebra interface based on the BLAS
- P1234: TODO: mdspan

Implementations

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

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

ISO C++ Day 2: Threads and Synchronization

AGENDA: part II

Threads and Synchronization

- Threads, Shared Memory, and Data-races
- Synchronization primitives: Atomics, Mutexes, Barriers

Lab 2: 2D heat equation

- Recap part 1
- Hiding communication with computation

Execution Policies & Forward Progress

Lab 3 (overview): Parallel Tree Construction

C++ Threads

Threads
Main thread

Every C++ program has a “main thread”…

…some time after startup, it calls the main function with the command line arguments.

C++11 allows programs to create more threads in a portable way…

```
int main(int argc, char* argv[]) {

   // ... do the thing ...

}
```

Threads
std::thread(Function, Args…)

std::thread constructor creates a new thread…

… that starts by calling a function with its arguments….

…and runs in parallel with the thread that started it…

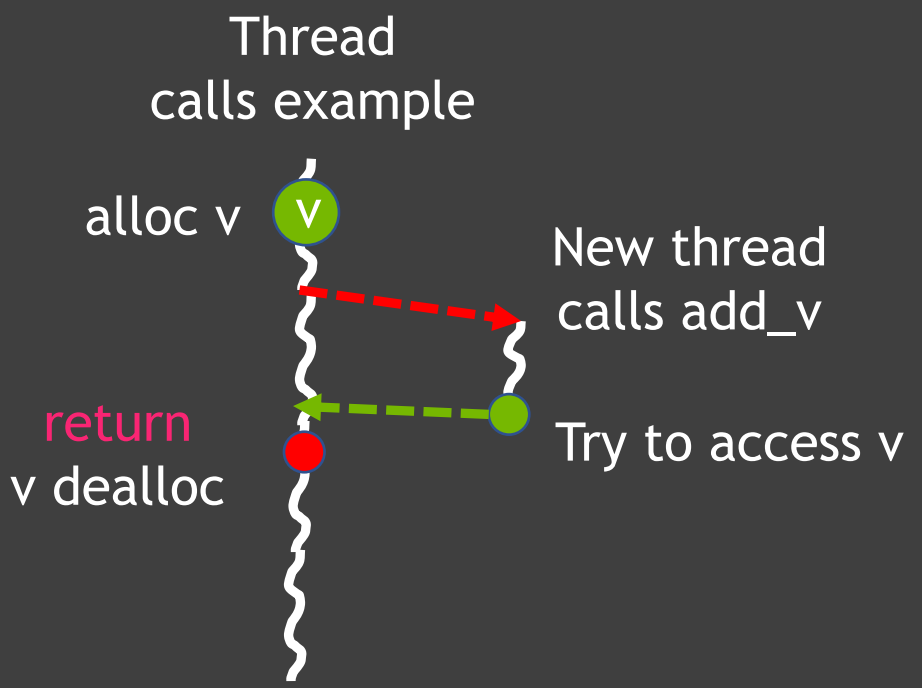
…and can outlive it!

```
#include <thread>

void example(int v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  // Overlapping computation
}
```

Try it on compiler-explorer!

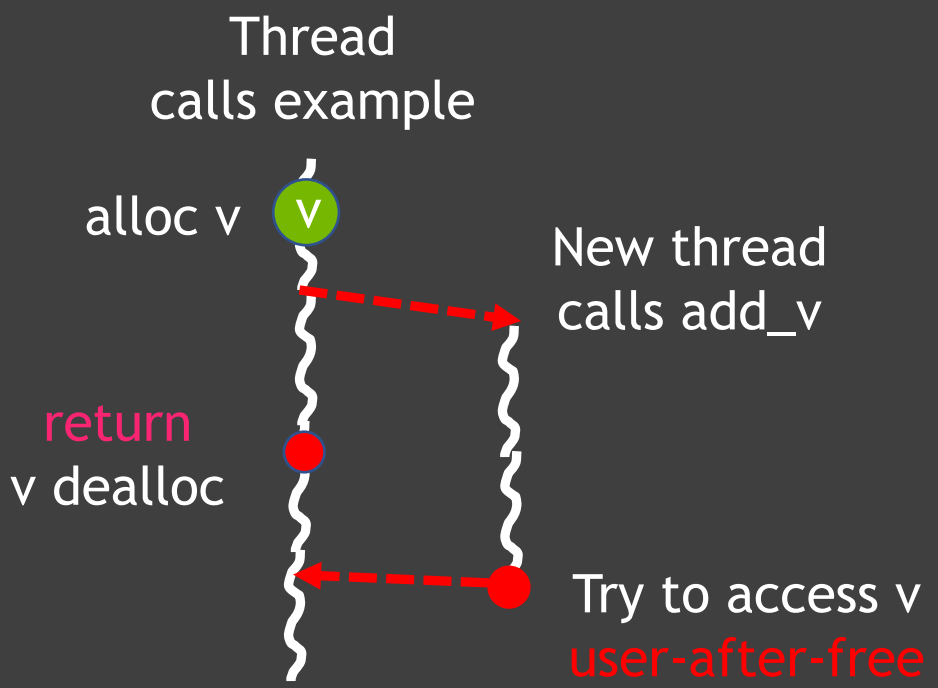
Threads
std::thread(Function, Args…)

```
#include <thread>

void caller() {
    example(13);
    // ...
}

void example(int v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  return;
}
```



Threads
std::thread(Function, Args…)

```
#include <thread>

void caller() {
   example(13);
   // ...
}

void example(int v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  return;
}
```



Threads
std::thread(Function, Args…)

std::thread prevents these accidental errors by exiting the program in its destructor (calls std::terminate)…

…this prevents the thread “t” from outliving the thread running “example”…

…this prevents the thread “t” from accessing “v” after it has been deallocated…

```
#include <thread>

void example(int v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  // Overlapping computation
}
```

```
Overlap work on thread 140648386959168
terminate called without an active exception
```

Threads
std::thread(Function, Args…)

std:: requires programs to handle this behavior explicitly:

1. t.join() waits until the thread completes
2. t.detach() allows t's thread to outlive t's destructor
3. std::jthread is a thread whose destructor calls t.join()
4. …we'll see more options in a while…

```
#include <thread>

void example(int v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  // Overlapping computation

  t.join();
}
```

Exercise: try all three options on compiler-explorer!

Shared Memory & Data Races

Shared memory

Until now only one thread modifies v…

…but what if both threads want to work on v in parallel ?

```
#include <thread>

void example(int& v) {
  auto add_v = [&](int a) { v += a; };

  auto t = std::thread(add_v, 42);

  add_v(8);

}
```

Shared memory Read-Modify-Write

A simple operation like:

```
v += 1;
```

is “shorthand notation” for:

```
tmp = *v; // Read to local
tmp += 1; // Modify local copy
*v = tmp; // Write local to memory
```

Shared memory and data races
Read-Modify-Write

A simple operation like:

```
v += 1;
```

is “shorthand notation” for:

```
tmp = *v; // Read to local
tmp += 1; // Modify local copy
*v = tmp; // Write local to memory
```

Example of two threads executing it with initially v = 0:

Thread 0

```
tmp = *v; // tmp==0
tmp += 1; // tmp==1
*v = tmp; // v==1
```

Thread 1

```
tmp = *v; // tmp==1
tmp += 1; // tmp==2
*v = tmp; // v==2
```

Afterwards:

v == 2

Shared memory Read-Modify-Write

A simple operation like:

```
v += 1;
```

is “shorthand notation” for:

```
tmp = *v; // Read to local
tmp += 1; // Modify local copy
*v = tmp; // Write local to memory
```

Another example of two threads executing it with initially v = 0:

Thread 0

```
tmp = *v; // tmp==0

tmp += 1; // tmp==1
*v = tmp; // v==1
```

Thread 1

```
tmp = *v; // tmp==0

tmp += 1; // tmp==1
*v = tmp; // v==1
```

Afterwards:

v == 1

Shared memory
Data races

C++ does not allow *concurrent unsynchronized* memory accesses to the *same memory* if *at least one is a write*.

Here, the two accesses here modify "v" without synchronization…

…C++ does not allow this, and this assert is not guaranteed to succeed.

```
int v = 0;

auto inc_v = [&] { v += 1; };


auto t0 = std::thread(inc_v);
inc_v();
t0.join();


assert(v == 2); // 0, 42, 2, anything!
```

Try it on compiler-explorer!

Why data races?
Compiler optimizations (cache data)

Loads a from memory every iteration, but in single-threaded programs a stays constant.

There we'd like to cache a locally to reduce memory traffic.

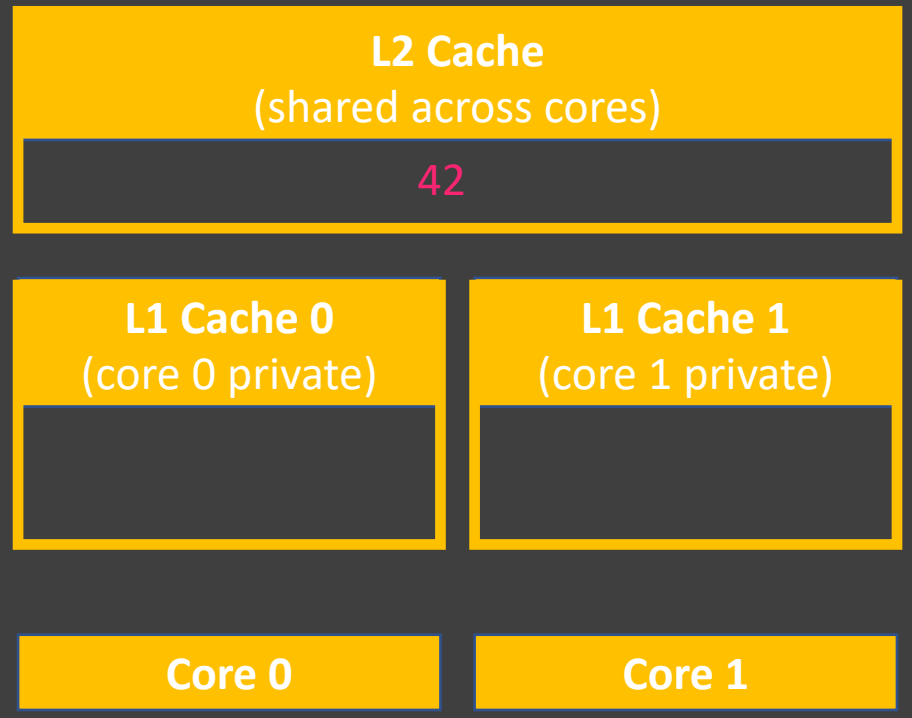
This (and many other) optimizations are not correct and very hard to do if threads are allowed to modify memory without synchronization.

```
void scale(int* y, int* a, int n) {
  assert(y != a);
  for (int i = 0; i < n; ++i) y[i] *= (*a);
}
```

```
int tmp = *a;
for (int i = 0; i < n; ++i) y[i] *= tmp;
```

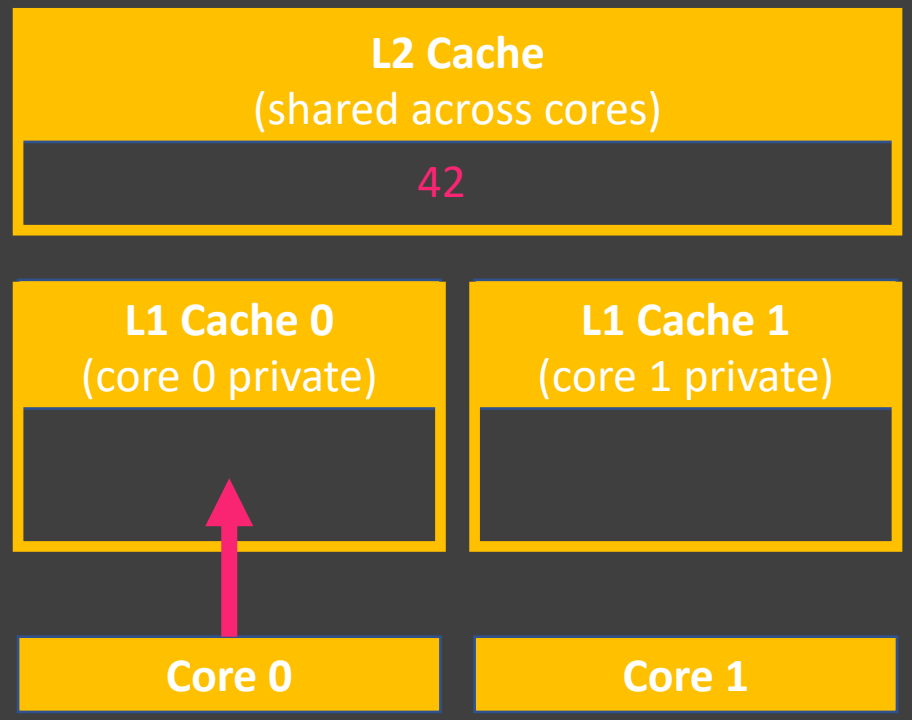
Why data races?
Hardware optimizations (cache data)

We have 42 stored at some address in memory…



Why data races?
Hardware optimizations (cache data)

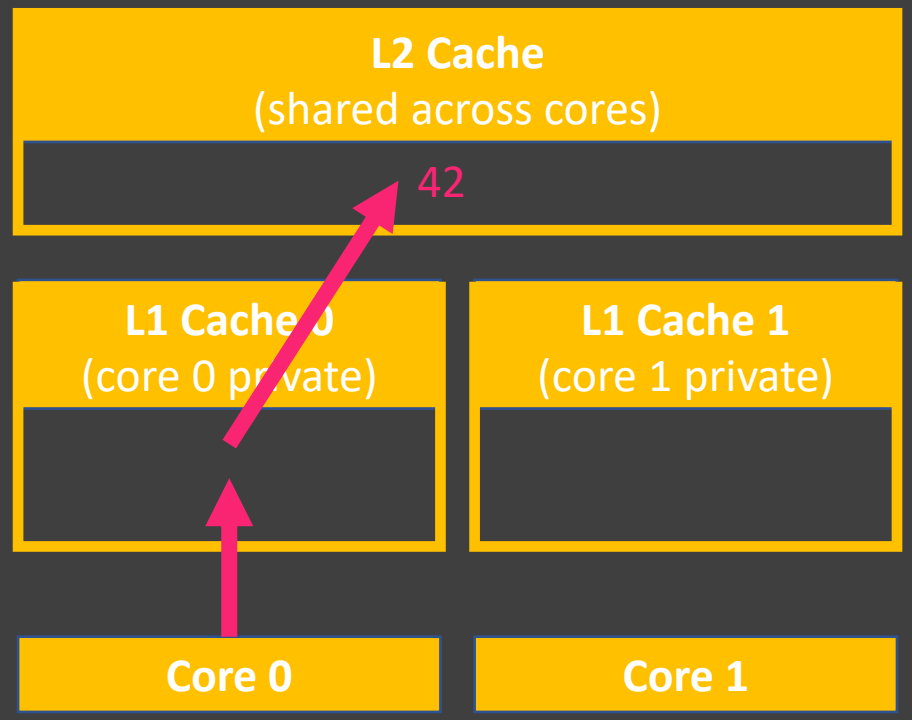
Core 0 reads from that address, and it misses on the L1 cache…



Why data races?
Hardware optimizations (cache data)

Core 0 reads from that address, and it misses on the L1 cache…

…request continues to the L2 cache where we find the value at that address…

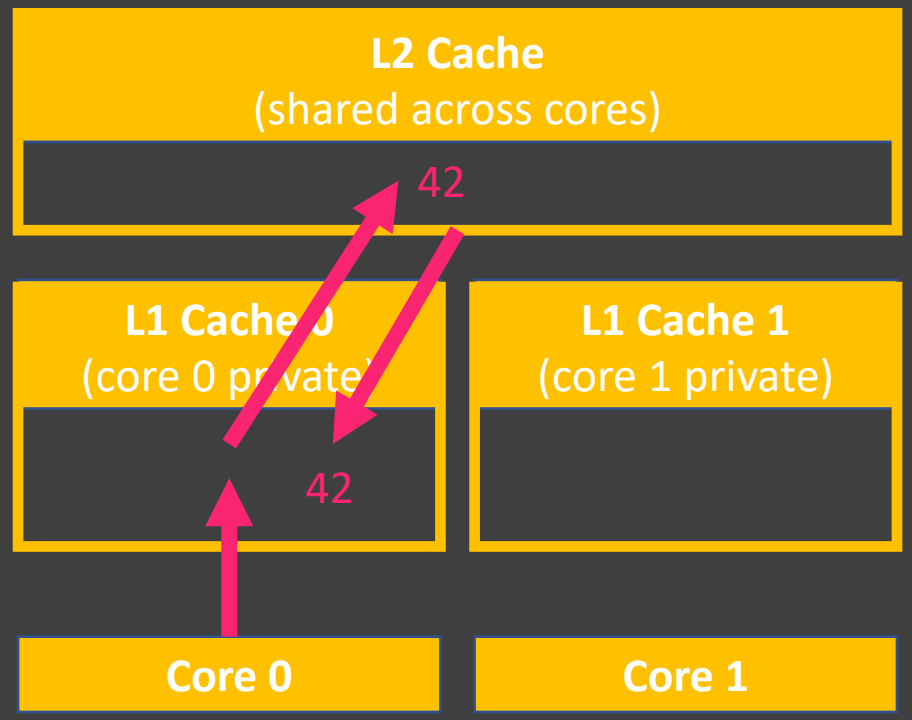


Shared memory and data races, why? Hardware optimizations (cache data)

Core 0 reads from that address, and it misses on the L1 cache…

…request continues to the L2 cache where we find the value at that address…

…so we bring it to the L1 for future luck…



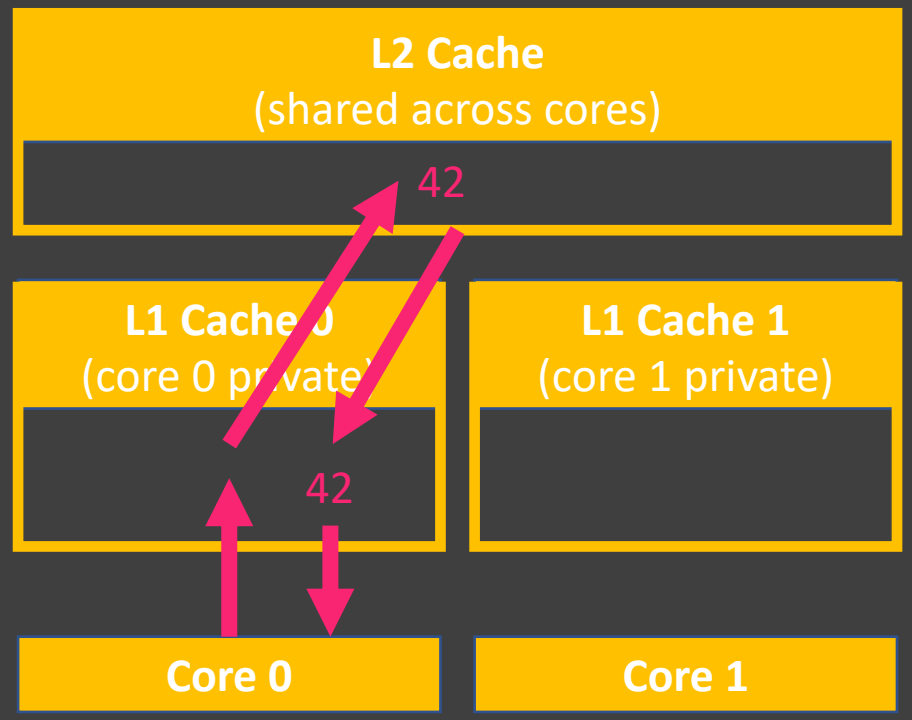
Why data races?
Hardware optimizations (cache data)

Core 0 reads from that address, and it misses on the L1 cache…

…request continues to the L2 cache where we find the value at that address…

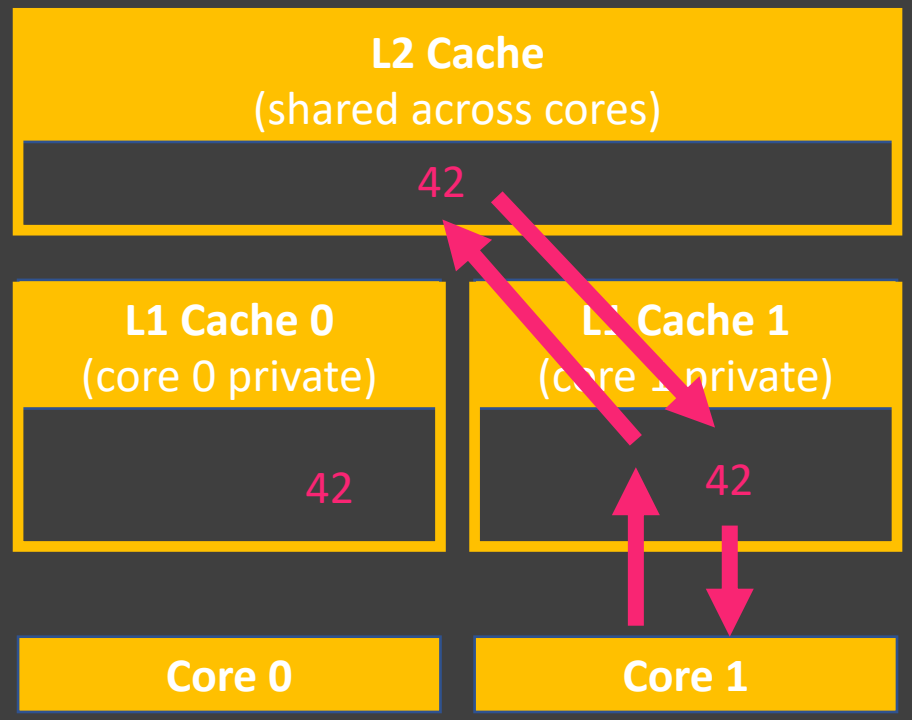
…so we bring it to the L1 for future luck…

…and finally to core 0 to compute.



Why data races?
Hardware optimizations (cache data)

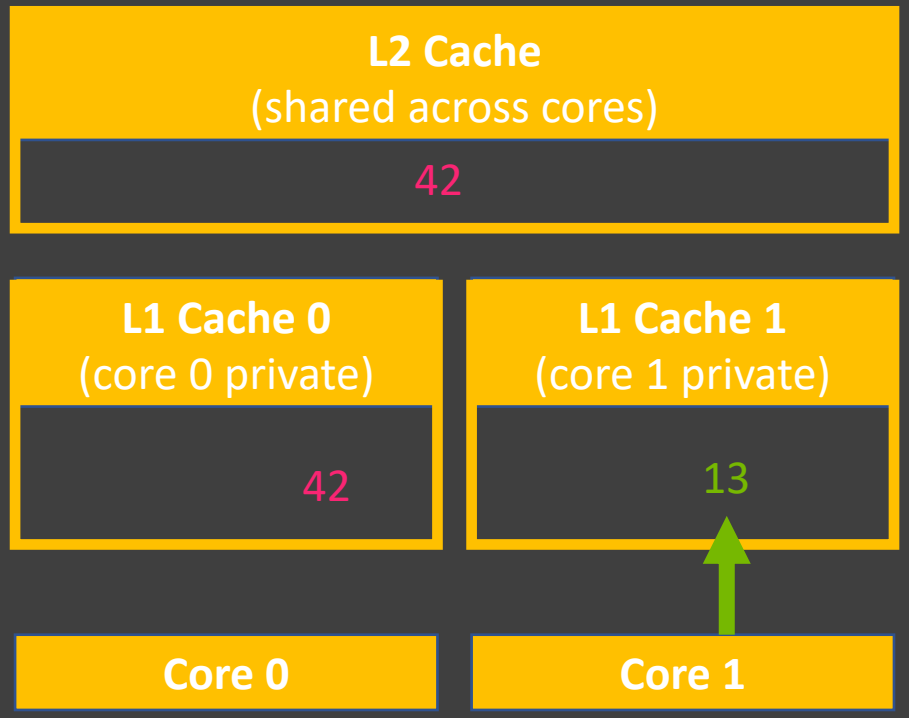
Core 1 does the same…



Why data races?
Hardware optimizations (cache data)

Core 1 does the same…

…but now Core 1 modifies it!

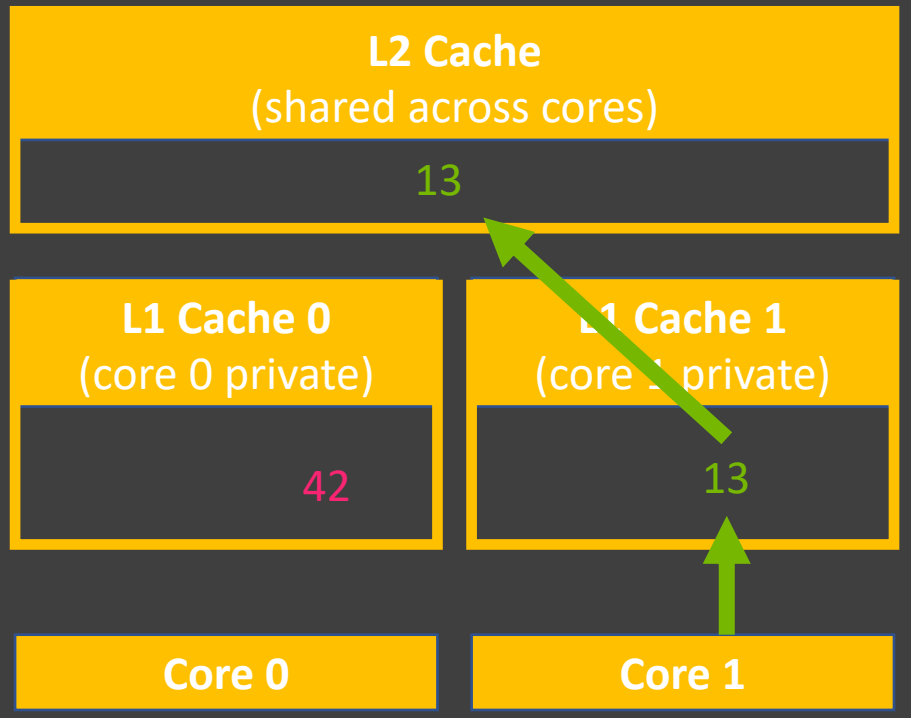


Why data races?
Hardware optimizations (cache data)

Core 1 does the same…

…but now Core 1 modifies it!

Core 1 waits for the modification to get to the L2…



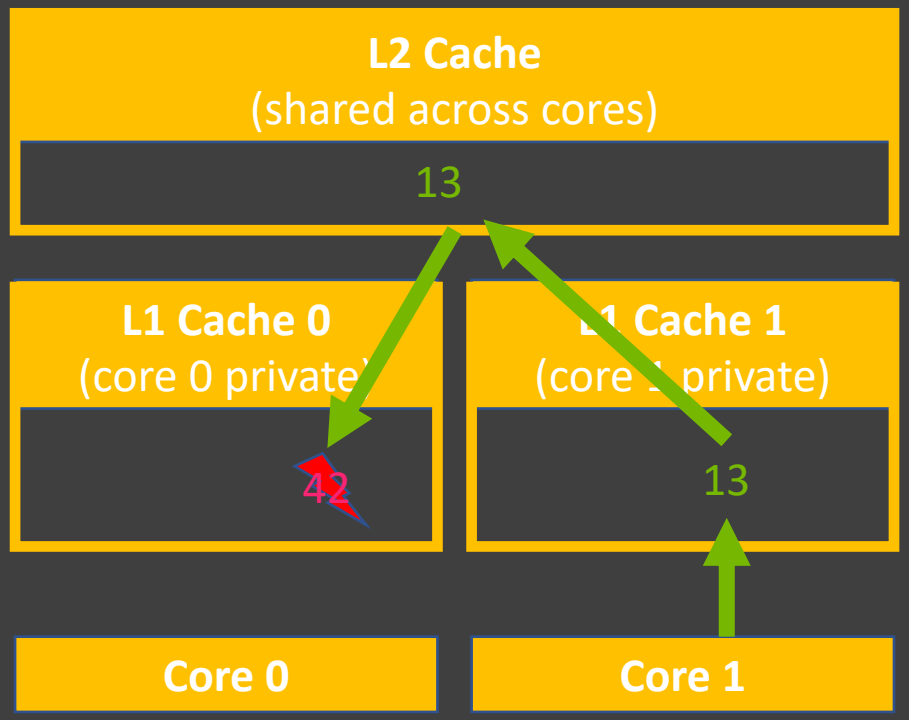
Why data races?
Hardware optimizations (cache data)

Core 1 does the same…

…but now Core 1 modifies it!

Core 1 waits for the modification to get to the L2…

Core 1 waits for the L2 to invalidate all other caches of all other cores…



Why data races?
Hardware optimizations (cache data)

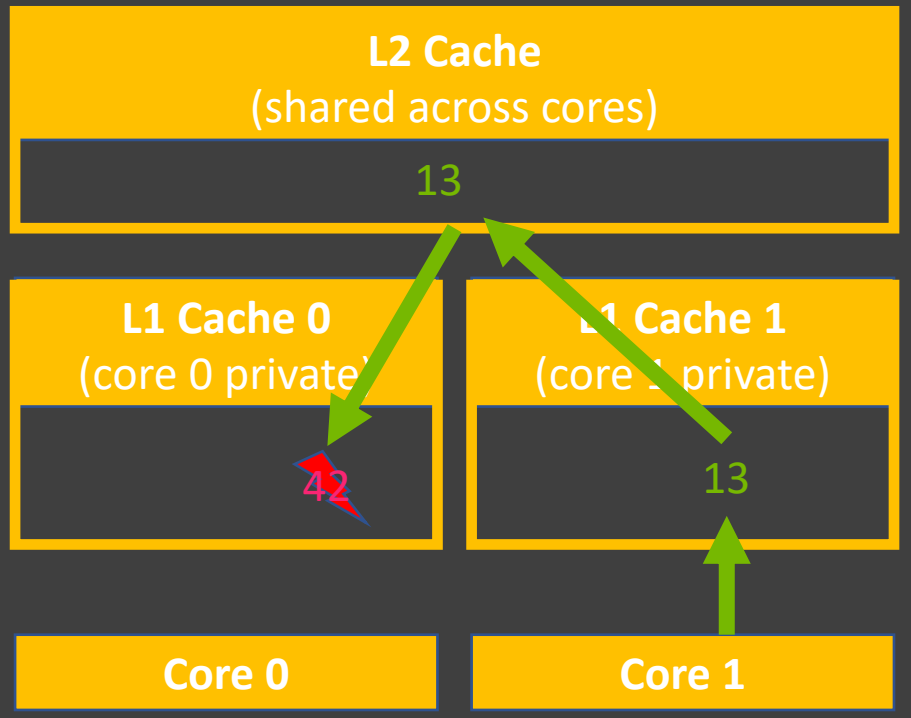
Core 1 does the same…

…but now Core 1 modifies it!

Core 1 waits for the modification to get to the L2…

Core 1 waits for the L2 to invalidate all other caches of all other cores…

…costs memory traffic!

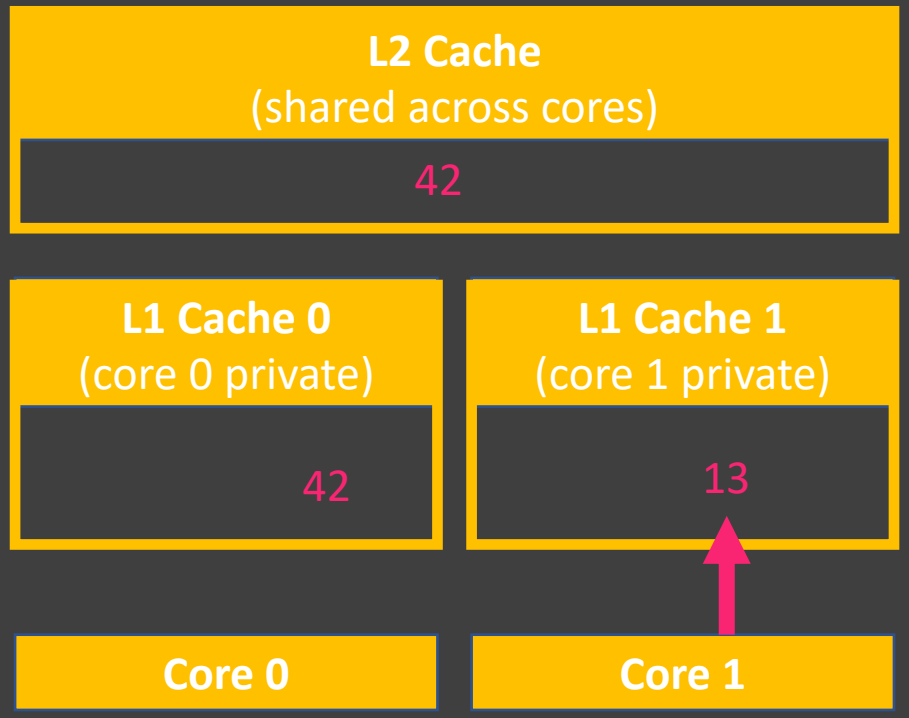


Why data races?
Hardware optimizations (cache data)

Hardware that assumes that if Core 1 modifies it no other Core will access it without synchronization is much faster…

…explicit instructions to “publish” modifications and ”invalidate” locally cached data…

…but requires programs to express when data must be published or reloaded…



Why data races?
Compiler and Hardware optimizations

C++ requires programs to express when to “publish” and “reload” modifications.

Forbidding data-races makes C++ programs that don’t do it “incorrect” and makes programs that are “correct” portable and fast.

Why data races?
Compiler and Hardware optimizations

C++ requires programs to express when to “publish” and “reload” modifications.

Forbidding data-races makes C++ programs that don’t do it “incorrect” and makes programs that are ”correct” portable and fast.

Compiler can optimize “scale” unless programmer expresses that “a” can be changed by a different thread.

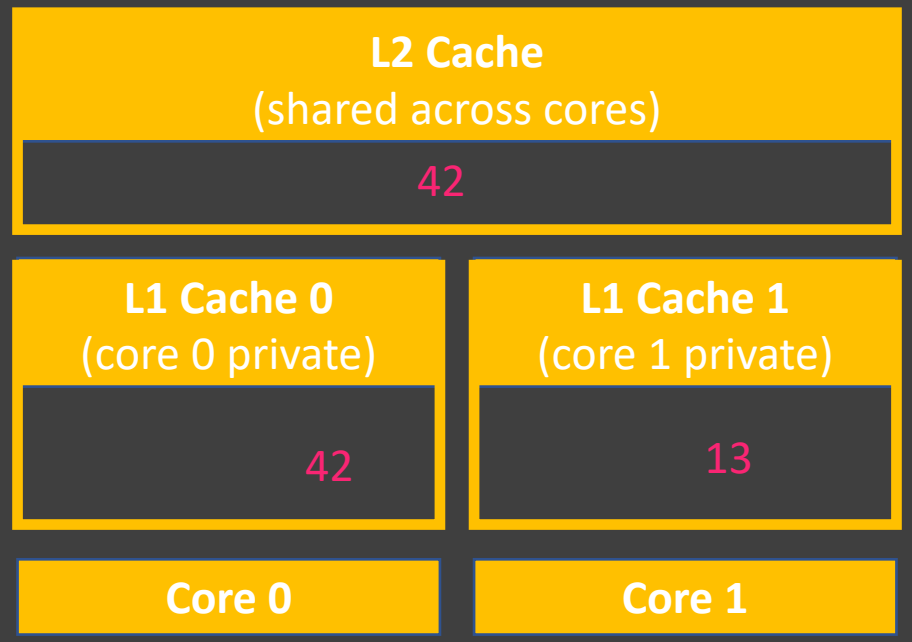
```
void scale(int* y, int* a, int n) {
  assert(y != a);
  int tmp = *a;
  for (int i = 0; i < n; ++i)
    y[i] += tmp;
}
```

Why data races?
Compiler and Hardware optimizations

C++ requires programs to express when to “publish” and “reload” modifications.

Forbidding data-races makes C++ programs that don’t do it “incorrect” and makes programs that are ”correct” portable and fast.

Vendors can build hardware that scales to millions of hardware threads.



Synchronization primitives: Atomics

Atomic variables
std::atomic

A wrapper over a value that ensures all operations performed on the value are atomic...

```
#include <atomic>

std::atomic<int> v = 0;

auto inc_v = [&] { v += 1; };

auto t0 = std::thread(inc_v);
inc_v();
t0.join();



assert(v == 2); // 2, Always!
```

Atomics references
std::atomic_ref

A wrapper over a reference that ensures all operations performed on the value referred to are atomic…

For the lifetime of the std::atomic_rer, all accesses to the referred value must happen through an std::atomic_ref object, but not necessarily the same one.

```
#include <atomic>

int w = 0;
std::atomic_ref v{w};

auto inc_v = [=v] { v += 1; };

auto t0 = std::thread(inc_v);
inc_v();
t0.join();



assert(v == 2); // 2, Always!
```

Atomic Memory Operations
std::atomic & std::atomic_ref

Atomic memory operations do not introduce data races when used from multiple threads…

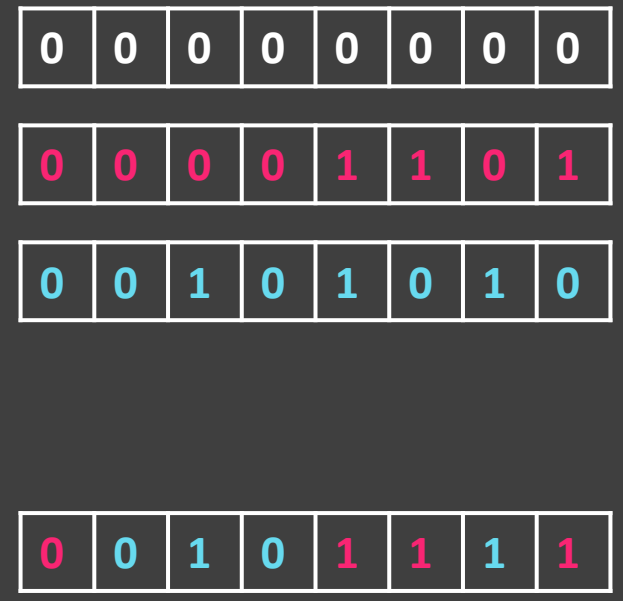
```
#include <atomic>

std::atomic<char> v = 0;

auto t = std::thread([&] { v = 42; });
v = 13;
t.join();

assert(v == 42 || v == 13);
```

Atomic Memory Operations
std::atomic & std::atomic_ref

Atomic memory operations do not introduce data races when used from multiple threads…

…and are indivisible!

Atomic operations are performed as a "unit" and C++ programs cannot observe partial results that mix data from independent operations.



Atomic Memory Operations
std::atomic & std::atomic_ref

Atomic memory operations do not introduce data races when used from multiple threads…

…are indivisible…

…and are fine-grained…

…allow programs to pay the cost of the precise synchronization (publishing / re-loading data) required only when its needed.

```
#include <atomic>

std::atomic<char> v = 0;
int w = 2;

auto t = std::thread([&] { v = w * 42; });
v = 13;
t.join();

assert(v == 42 || v == 13);
```

Atomic Memory Operations
std::atomic & std::atomic_ref

Exercise: fix example from previous section using std::atomic on compiler-explorer!

Exercise: fix example from previous section using std::atomic_ref on compiler-explorer!

Synchronization primitives: Mutex

Synchronizing access to multiple memory locations

Programs often need to synchronize access to multiple memory locations like “v” and “w”…

…this program has data-races and is incorrect…

…C++ does not guarantee anything about what it does.

```
char v = 0, w = 0;

auto mod = [&](char x, char y) {
    v = x; w = y;
};

auto t = std::thread(mod, 42, 13);
mod(13, 42);
t.join();

assert(
  (v==42 && w==13) || (v==13 && w==42)
);
```

Synchronizing access to multiple memory locations

We can remove the data-races by using atomic memory operations instead…

…however, we want the program to only produce these two outcomes, but it can produce more…

```
std::atomic<char> v = 0, w = 0;

auto mod = [&](char x, char y) {
    v = x; w = y;
};

auto t = std::thread(mod, 42, 13);
mod(13, 42);
t.join();

assert(
  (v==42 && w==13) || (v==13 && w==42)
);
```

Synchronizing access to multiple memory locations

```
std::atomic<char> v = 0, w = 0;

auto mod = [&](char x, char y) {
    v = x; w = y;
};

auto t = std::thread(mod, 42, 13);
mod(13, 42);
t.join();

assert(
  (v==42 && w==13) || (v==13 && w==42)
);
```

Example of two threads executing it with initially v = 0, w = 0:

Thread 0	Thread 1
v = 13;	
w = 42;	
	v = 42;
	w = 13;

Afterwards:

v == 42, w == 13

Synchronizing access to multiple memory locations

```
std::atomic<char> v = 0, w = 0;

auto mod = [&](char x, char y) {
    v = x; w = y;
};

auto t = std::thread(mod, 42, 13);
mod(13, 42);
t.join();

assert(
  (v==42 && w==13) || (v==13 && w==42)
);
```

Another example of two threads executing it with initially v = 0, w = 0:

Thread 0	Thread 1
`v = 13;`	
	`v = 42;`
	`w = 13;`
`w = 42;`	

Afterwards:

v == 42, w == 42

Mutual Exclusion – Critical sections std::mutex

std::lock_guard locks the mutex producing a guard that unlocks it on destruction.

Only 1 thread helds the mutex any time.

Accesses to data protected by the mutex - i.e. only accessed while mutex is held - avoid data races (happen from only 1 thread at a time).

Exercise: fix this example!

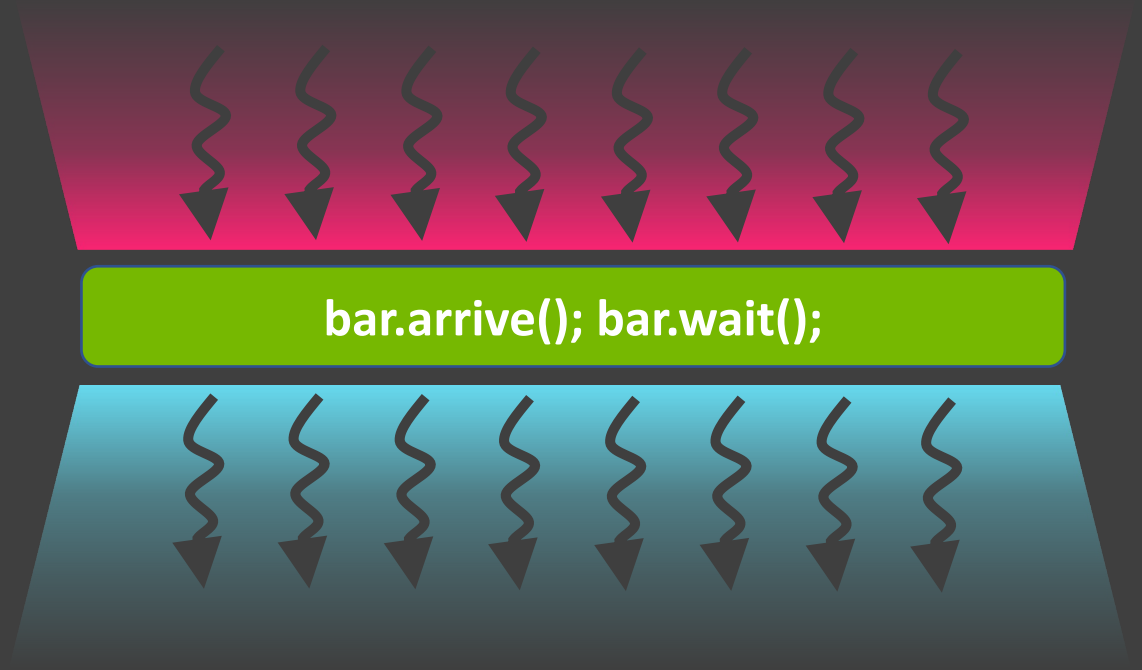
```
#include <mutex>
std::mutex m;
char v = 0, w = 0;

auto mod = [&](char x, char y) {
    auto g = std::lock_guard{m};
    v = x; w = y;
};

auto t = std::thread(mod, 42, 13);
mod(13, 42);
t.join();
assert( // Always!
  (v==42 && w==13) || (v==13 && w==42)
);
```

Synchronization primitives: Barrier

Producer / consumer synchronization
std::barrier

std::barrier blocks threads at "wait" until "expected_arrivals" threads have "arrived"…

```
#include <barrier>
std::barrier bar(expected_arrivals);
```



Producer / consumer synchronization
std::barrier

std::barrier blocks threads at “wait” until “expected_arrivals” threads have “arrived”…

…and synchronizes memory…

…making all memory operations made by individual threads before “arrive” visible to all threads unblocked from “wait”.

```
#include <barrier>
std::barrier bar(expected_arrivals);
```

Memory operations by threads before “arrive”…

…are visible to threads unblocked from “wait”.

Producer / consumer synchronization std::barrier

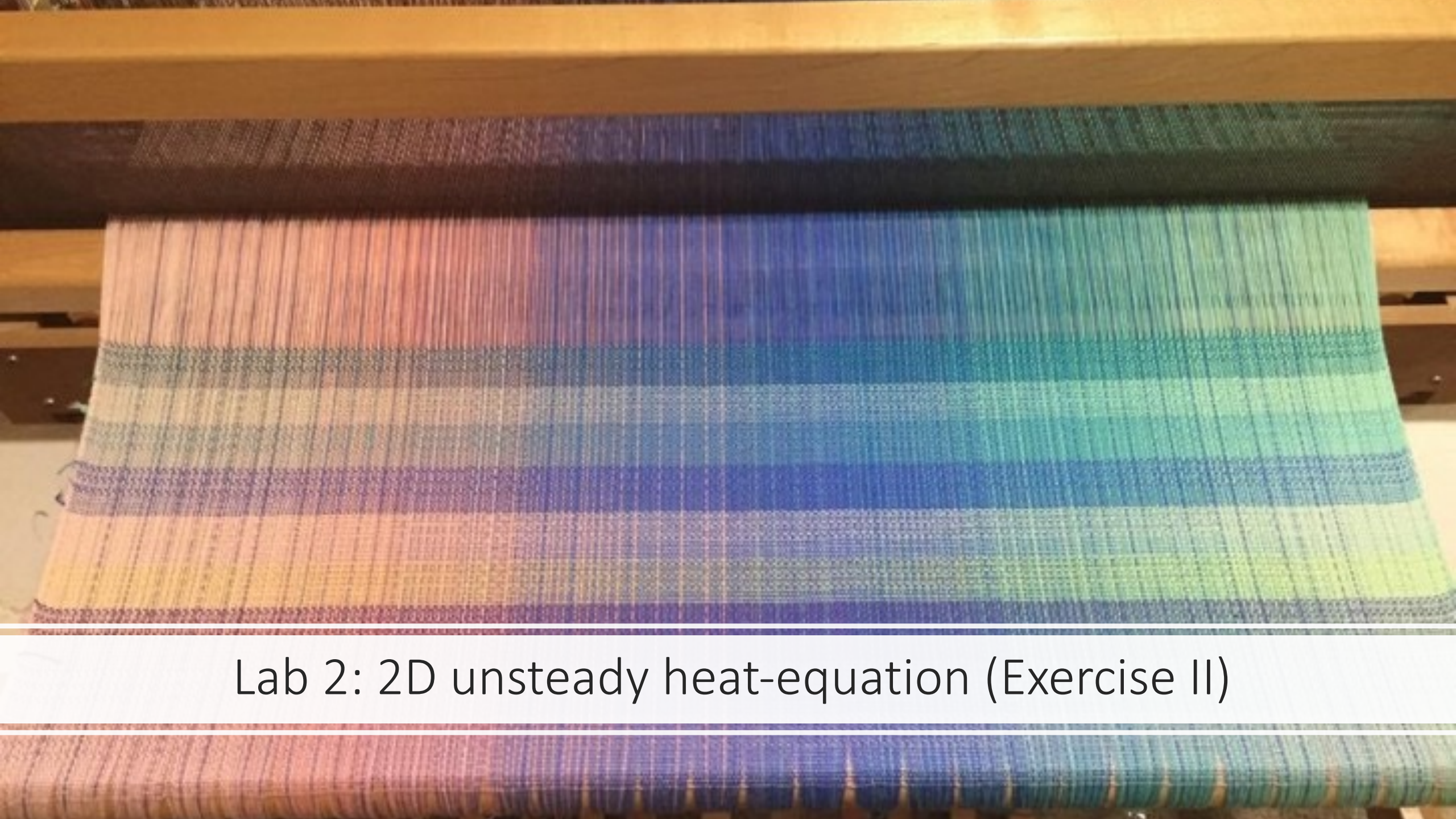
Thread 0 produces “v0”, and thread 1 produces “v1”…

Thread 0 “arrives” at the barrier to signal that “v0” is ready…

Thread 1 “arrives and waits” at the barrier to signal that v1 is ready, and “wait” until ”v0” becomes ready too.

Exercise: fix this example!

```
#include <barrier>
std::barrier b(2);

int v0 = 0, v1 = 0;

std::thread t0([&] {
  v0 = std::thread::get_id();
  b.arrive();
});
t0.detach();

v1 = std::thread::get_id();
b.arrive_and_wait();
print(v0, v1);
```

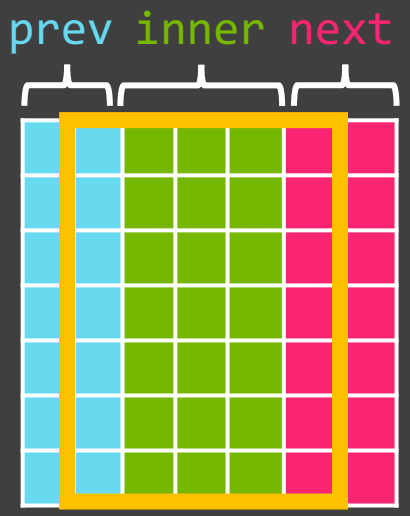


Lab 2: 2D unsteady heat-equation (Exercise II)

2D unsteady heat equation
Recap?

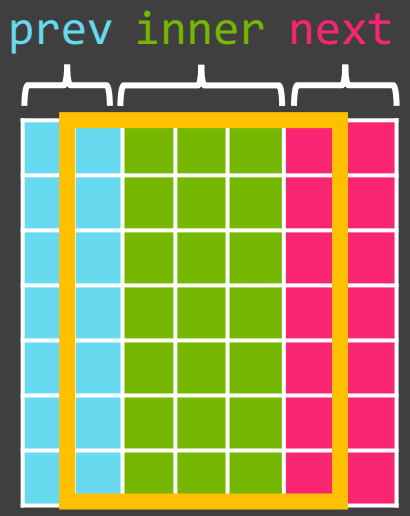
2D unsteady heat equation: Implementation

```
for (int it = 0; it < niterations; ++it) {
    float e_prev = []{ exchange(prev); stencil(prev); }();
    float e_next = []{ exchange(next); stencil(next); }();
    float e_inner = stencil(inner);
    float e = e_prev + e_next + e_inner;
    MPI_Reduce(&e, …);
    if (rank == 0) print(e);
}
```



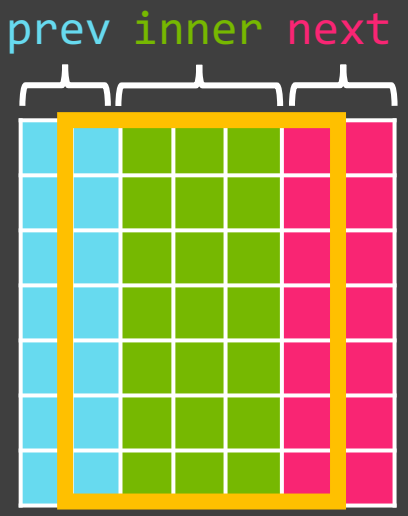
2D unsteady heat equation: Iteration Timeline

```
for (int it = 0; it < niterations; ++it) {
    float e_prev = []{ exchange(prev); stencil(prev); }();
    float e_next = []{ exchange(next); stencil(next); }();
    float e_inner = stencil(inner);
    float e = e_prev + e_next + e_inner;
    MPI_Reduce(&e, …);
    if (rank == 0) print(e);
}
```



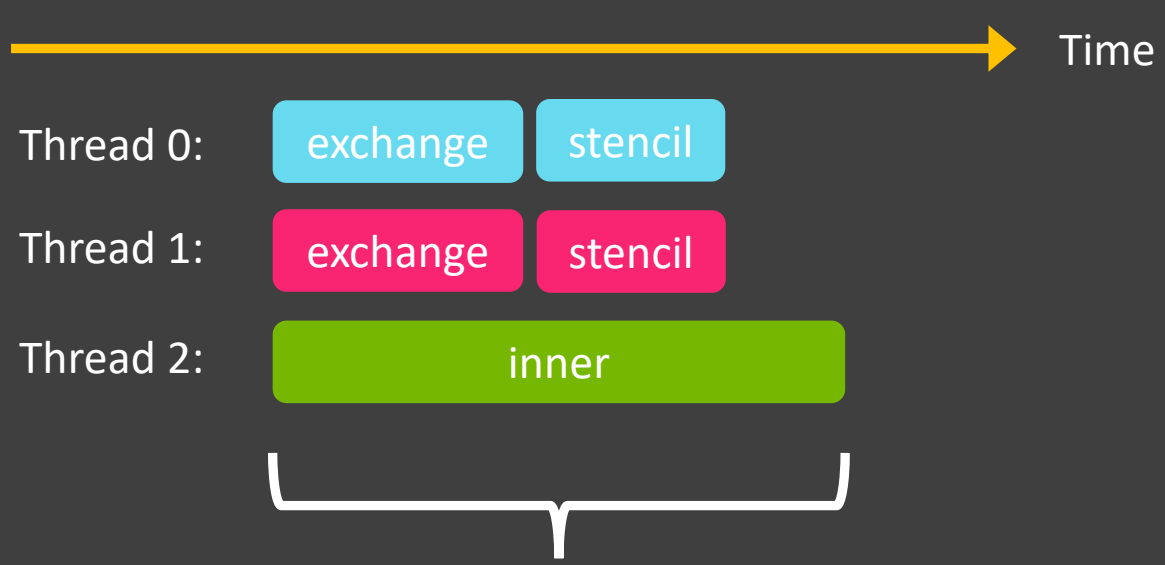
2D unsteady heat equation: Iteration Timeline

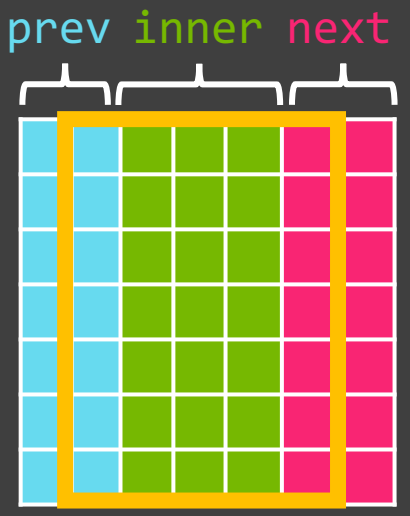
```
for (int it = 0; it < niterations; ++it) {
    float e_prev = []{ exchange(prev); stencil(prev); }();
    float e_next = []{ exchange(next); stencil(next); }();
    float e_inner = stencil(inner);
    float e = e_prev + e_next + e_inner;
    MPI_Reduce(&e, …);
    if (rank == 0) print(e);
}
```



$$T_{\text{execution}} = T_{\text{exchange}} + T_{\text{boundary}} + T_{\text{inner}}$$

2D unsteady heat equation: 3 threads



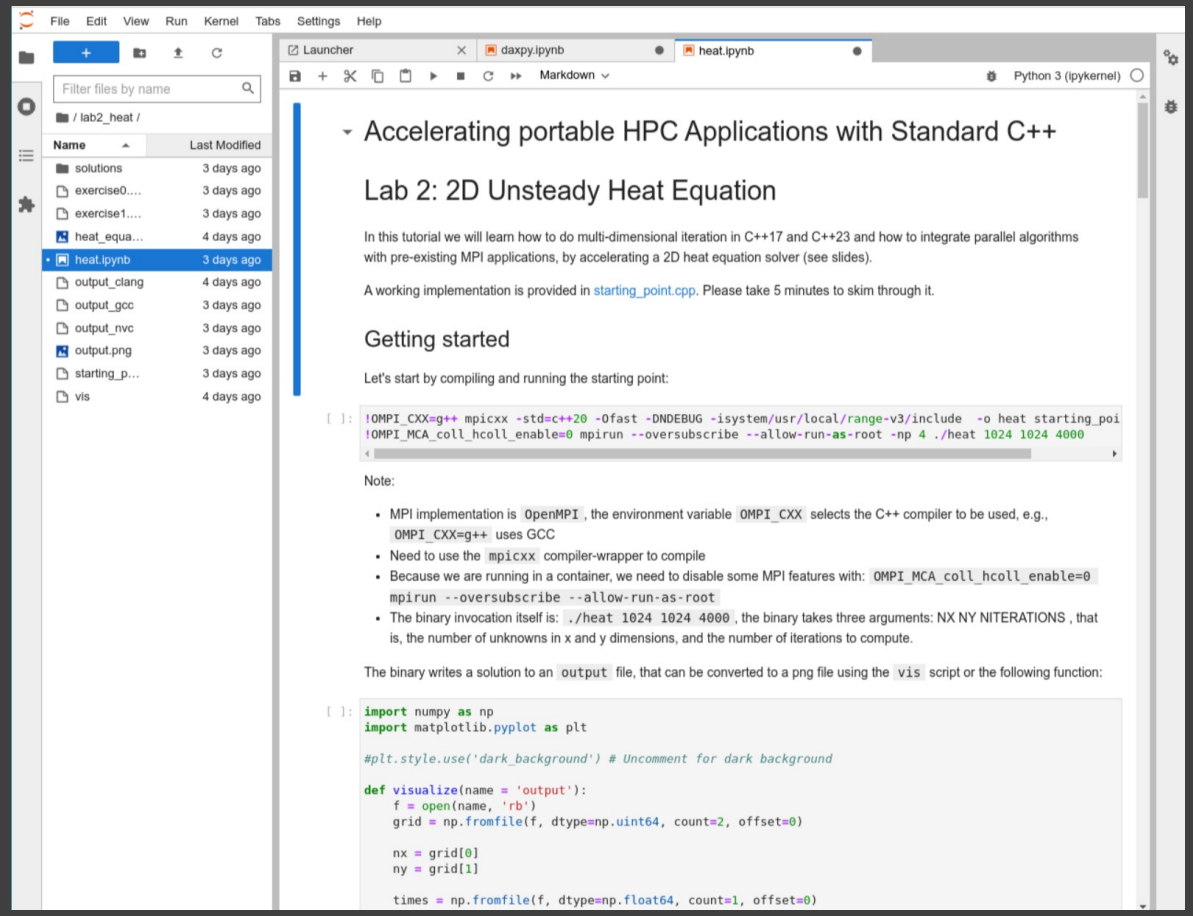


$$T_{\text{execution}} = \max(T_{\text{inner}}, T_{\text{exchange}} + T_{\text{boundary}})$$

Lab 2: 2D heat equation (Part II)

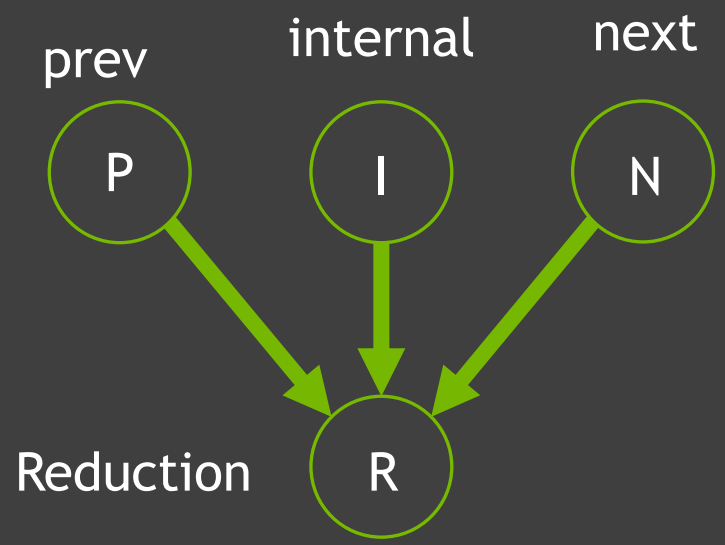
- **Exercise 1:** parallelize the MPI implementation using the STL parallel algorithms with the different types of indexing to create an hybrid MPI/C++ application.

- **Exercise 2:** hide communication behind computation using std::thread, std::atomic, and std::barrier



With std::thread, std::atomic, and std::barrier

```
std::atomic<double> energy{0.};
std::barrier bar(3);

auto internal = …;
auto t_prev = std::thread(prev_boundary);
auto t_next = std::thread(next_boundary);
auto t_internal = std::thread(internal);

t_prev.join();
t_next.join();
t_internal.join();
```



Each thread processes all iterations
Using std::barrier to synchronize

```
auto internal = [&] {
  for (long it = 0; it < p.nit(); ++it) {
    energy += internal(u_new, u_old, p);
    bar.arrive_and_wait();
    double e = energy.load();
    MPI_Reduce(&e, &e, 1,…, MPI_SUM,…);
    energy.store(0.);
    bar.arrive_and_wait();
    std::swap(u_new, u_old);
  }
};
auto t_internal = std::thread(internal);
```

```
auto prev = [&] { // and next
  for (long it = 0; it < p.nit(); ++it) {
    energy += prev_boundary(u_new, u_old, p);
    bar.arrive_and_wait();
    bar.arrive_and_wait();
    std::swap(u_new, u_old);
  }
};

auto t_internal = std::thread(prev);
```

MPI Reduction done in one thread inside a critical section

```
auto internal = [&] {
  for (long it = 0; it < p.nit(); ++it) {
    energy += internal(u_new, u_old, p);
    bar.arrive_and_wait();
    double e = energy.load();
    MPI_Reduce(&e, &e, 1,…, MPI_SUM,…);
    energy.store(0.);
    bar.arrive_and_wait();
    std::swap(u_new, u_old);
  }
};
auto t_internal = std::thread(internal);
```

Lab 2: 2D heat equation (Part II)
Solutions (DEMO)

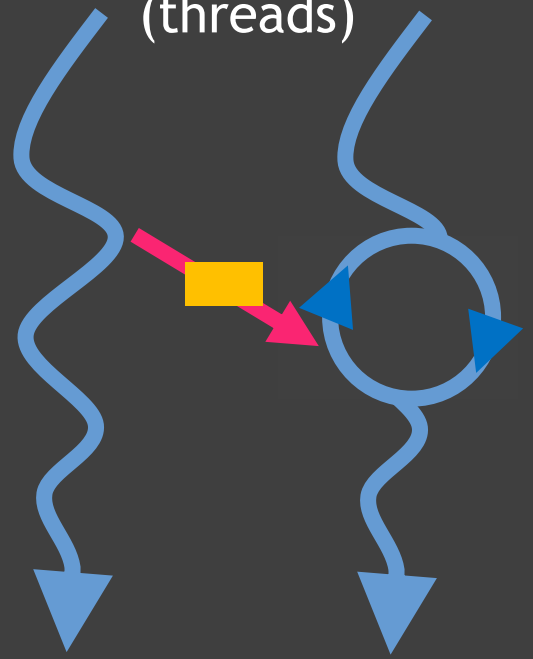
With P2300 std::execution
No atomics & no barriers in user code!

```
stde::sender auto solver(stde::scheduler auto sch, …) {
  auto prev  = stde::schedule(sch) | stde::then([…]{ return prev_boundary)(…) };
  auto next  = stde::schedule(sch) | stde::then([…]{ return next_boundary)(…) };
  auto inner = stde::schedule(sch) | stde::then([…]{ return internal(…);      });

  auto step = stde::when_all(prev, next, inner)
    | stde::then([](double e0, double e1, double e2) {
        double e = e0 + e1 + e2;
        MPI_Reduce(&e, &e, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    });
  return step | repeat_effect_until([nit] mutable { return --nit == 0; });
}


static_thread_pool ctx{3};
std::this_thread::sync_wait(std::move(solver(ctx.get_scheduler())));
```

Execution Policies & Forward Progress

ISO C++ Execution Policies

Parallel Execution Policy
`std::execution::par`
(threads)

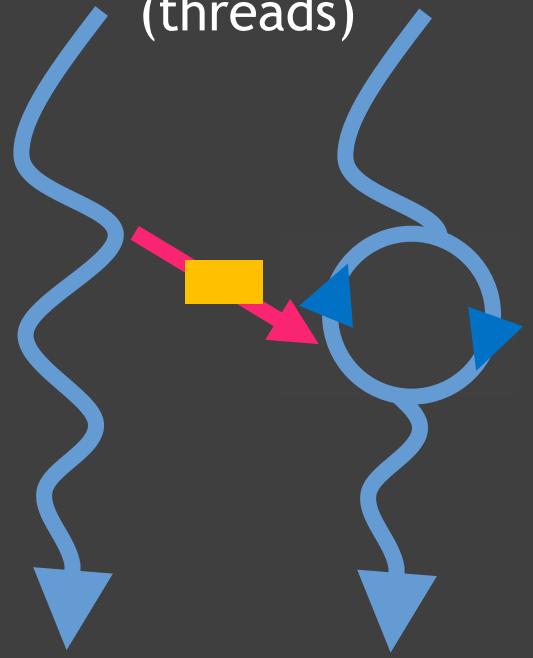


Starvation Free Algorithms

Threads may wait for messages

ISO C++ Execution Policies

Parallel Execution Policy
`std::execution::par`
(threads)



Starvation Free Algorithms

Threads may wait for messages

Parallel Unsequenced Execution Policy
`std::execution::par_unseq`
(SIMD lanes)

Independent Execution

Threads shall not exchange messages

ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
int global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
int global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
int global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

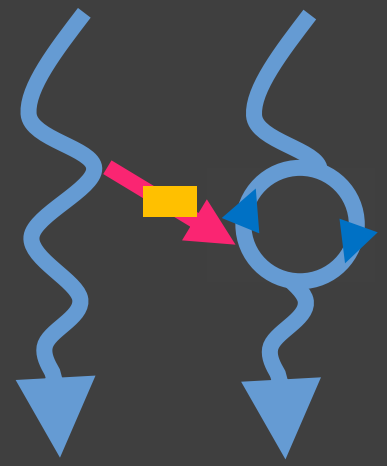
ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```



ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

ISO C++ Execution Policies

Parallel Execution Policy
`par` (threads)

```
int global;
str::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

Parallel Unsequenced Execution Policy
`par_unseq` (SIMD lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

```
int global = std::reduce(par_unseq, b, e,
                         std::plus<>{});
```

ISO C++ Execution Policies Hardware support

Parallel Execution Policy

`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

- Starvation-free algorithms

Parallel Unsequenced Execution Policy

`par_unseq` (SIMD Lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

- Independent computation

ISO C++ Execution Policies Hardware support

Parallel Execution Policy

`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

- Starvation-free algorithms
- Fair scheduler

Parallel Unsequenced Execution Policy

`par_unseq` (SIMD Lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

- Independent computation
- Any scheduler

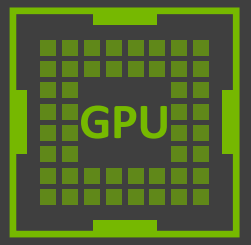
ISO C++ Execution Policies Hardware support

Parallel Execution Policy

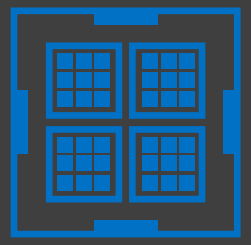
`par` (threads)

```
int global;
std::for_each(par, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

- Starvation-free algorithms
- Fair scheduler



NVIDIA GPU Threads



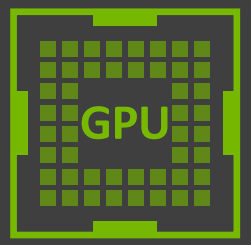
CPU Threads

Parallel Unsequenced Execution Policy

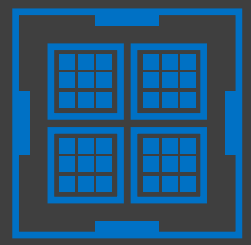
`par_unseq` (SIMD Lanes)

```
std::atomic<int> global;
std::for_each(par_unseq, b, e, [&](int& x) {
    auto g = std::lock_guard{mutex};
    global += x;
});
```

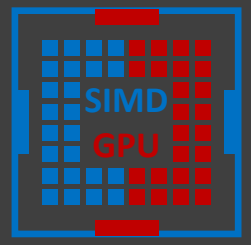
- Independent computation
- Any scheduler



NVIDIA GPU Threads



CPU Threads and SIMD



Other GPU Threads

ISO C++ Element Access Functions Summary

Element Access Functions must satisfy the requirements of the execution policy:

- par: data-race freedom
- par_unseq: independent computation

Functions used by parallel algorithms to access elements:

- Iterator Category operations (it++)
- Operations required by algorithm (std::sort calls std::iter_swap)
- User provided operations (lambdas)
- Specified uses of user provided operations (algorithm calls the lambda)

Programmer *guarantees* that Element Access Functions satisfy Execution Policy requirements.

Compiler and hardware *assume* that this is the case.

ISO C++ Execution Policies requirements

Execution Policy requirements on algorithm operations:

- **par**: data-race freedom
- **par_unseq**: independent

ISO C++ Execution Policies requirements on Element Access Functions

Execution Policy requirements on Element Access Functions:

- par: data-race freedom
- par_unseq: independent

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}

// [1, 2, 3] -> [2, 4, 6] -> 12
v = std::transform_reduce(b, e, 0., std::plus<>{},
                                    [](int i) { return i * 2; });
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}
```

Copies and inserts elements [b, e) into the vector “v”

```
std::vector v;
std::copy(b, e, std::back_inserter(v));
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations

Requirements:

- par: not data-race free
- par_unseq: not independent

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}

std::vector v;
std::copy(b, e, std::back_inserter(v));
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm

O(e - b) applications each of reduce and transform

Other algorithms like, e.g., std::sort call std::iter_swap.

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}

// [1, 2, 3] -> [2, 4, 6] -> 12
v = std::transform_reduce(b, e, 0., std::plus<>{},
                                    [](int i) { return i * 2; });
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}

// [1, 2, 3] -> [2, 4, 6] -> 12
v = std::transform_reduce(b, e, 0., std::plus<>{},
                                    [](int i) { return i * 2; });
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

```
T transform_reduce(It b, It e, T init, Red r, Map m) {
    for (; b != e; ++b)
        init = r(init, m(*b));
    return init;
}

// [1, 2, 3] -> [2, 4, 6] -> 12
v = std::transform_reduce(b, e, 0., std::plus<>{},
                                    [](int i) { return i * 2; });
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

```
int max = std::numeric_limits<int>::min();
v = std::transform_reduce(par, b, e, 0., plus<>{},
                                  [](int i) {
    max = std::max(max, i);
    return i * 2;
});
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

```
int max = std::numeric_limits<int>::min();
v = std::transform_reduce(par, b, e, 0., plus<>{},
                                  [](int i) {
    max = std::max(max, i); // UB: data-race
    return i * 2;
});
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

```
atomic<int> max = std::numeric_limits<int>::min();
v = std::transform_reduce(par, b, e, 0., plus<>{},
                                  [](int i) {
    max.fetch_max(i); // OK
    return i * 2;
});
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

```
atomic<int> max = std::numeric_limits<int>::min();
v = std::transform_reduce(par_unseq, b, e, 0.,plus<>{},
                                []( int i) {
    max.fetch_max(i); // UB: not independent
    return i * 2;
});
```

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

Must satisfy the requirements of the execution policy:

- par: data-race freedom
- par_unseq: independent computation

ISO C++ Element Access Functions

Functions used by parallel algorithms to access elements:

- Iterator Category operations
- Operations required by algorithm
- User provided operations
- Specified uses of user provided operations

Must satisfy the requirements of the execution policy:

- par: data-race freedom
- par_unseq: independent computation

Programmer *guarantees* that Element Access Functions satisfy Execution Policy requirements.

Compiler and hardware *assume* that this is the case.

Lab 3: Parallel tree construction

Trie: data-structure to accelerate searches

Trie data-structures accelerate searches.

Like `std::map<std::string, something>`.

Example: #word occurrences in book collection.

Build a `std::map<std::string, count>`.

Trie: data-structure to accelerate searches

Trie data-structures accelerate searches.

Like `std::map<std::string, something>`.

Example: #word occurrences in book collection.

Build a `std::map<std::string, count>`.

```
input = ["to","inn","tea","to","ten","A","ted","in"]
```

Trie: data-structure to accelerate searches

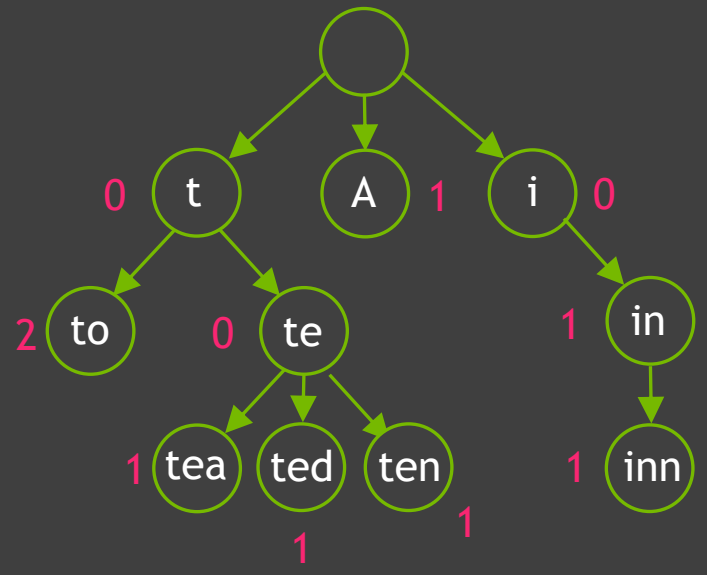
Trie data-structures accelerate searches.

Like `std::map<std::string, something>`.

Example: #word occurrences in book collection.

Build a `std::map<std::string, count>`.

```
input = ["to","inn","tea","to","ten","A","ted","in"]
```



Trie: data-structure to accelerate searches

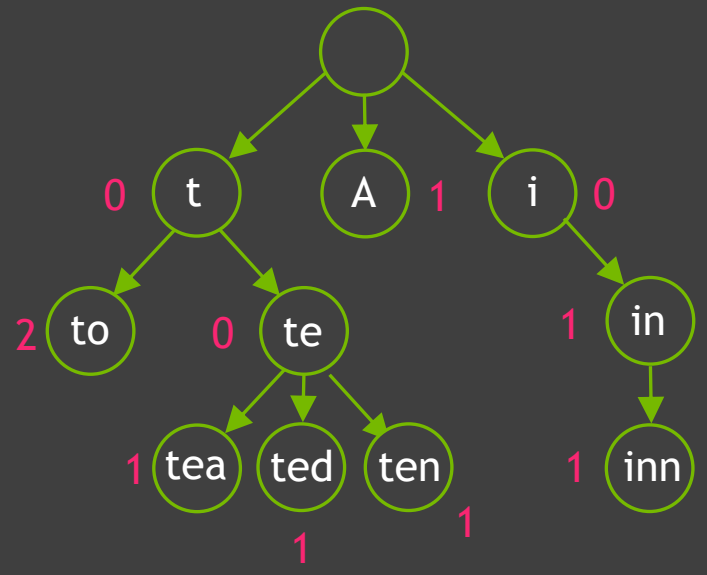
Trie data-structures accelerate searches.

Like `std::map<std::string, something>`.

Example: #word occurrences in book collection.

Build a `std::map<std::string, count>`.

```
input = ["to","inn","tea","to","ten","A","ted","in"]
```




```
struct trie {
    struct child_ref {
        trie* ptr;
    };
    std::array<child_ref, 26> next; // edges to children
    int count;                      // node value
};
int idx_of(char c) {
    if (c >= 'a' && c <= 'z') return c - 'a';
    if (c >= 'A' && c <= 'Z') return c - 'A';
    return -1;  // other chars are delimiters
}
```

Trie: data-structure to accelerate searches

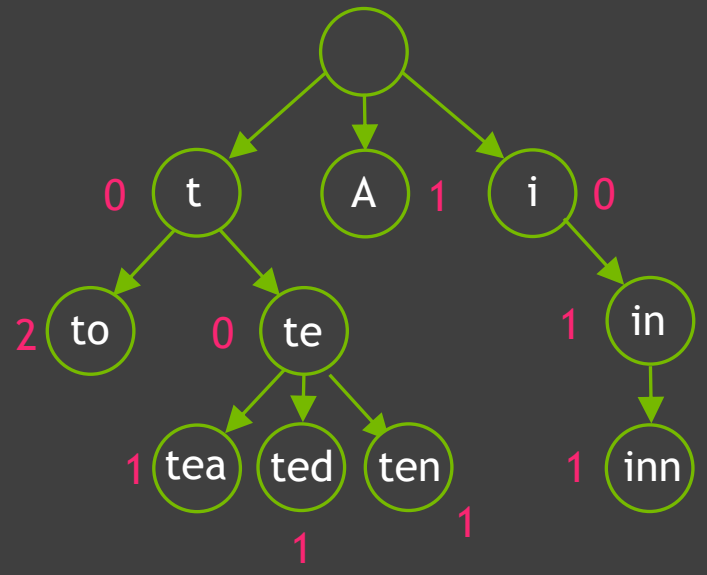
Trie data-structures accelerate searches.

Like `std::map<std::string, something>`.

Example: #word occurrences in book collection.

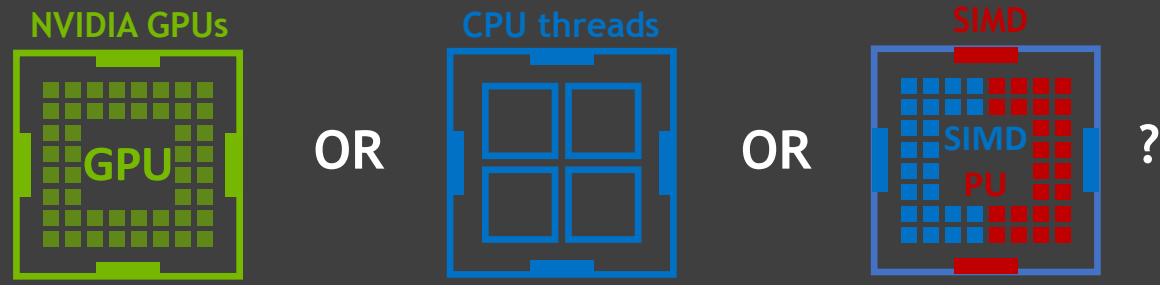
Build a `std::map<std::string, count>`.

```
input = ["to","inn","tea","to","ten","A","ted","in"]
```




```
struct trie {
    struct child_ref {
        trie* ptr;
    };
    std::array<child_ref, 26> next; // edges to children
    int count;                      // node value
};
int idx_of(char c) {
    if (c >= 'a' && c <= 'z') return c - 'a';
    if (c >= 'A' && c <= 'Z') return c - 'A';
    return -1;  // other chars are delimiters
}
```

- Building a node-based data structure.
- **Which execution resource would you pick and why?**



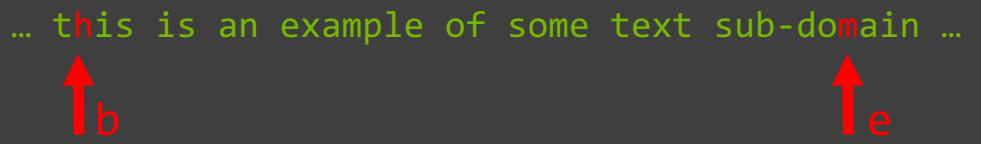
Trie append Prelude

```
void append(trie& root, trie*& bump, char const* begin, char const* end, unsigned domain, unsigned domains) {
    auto const size = end - begin;
    auto const domain_size = size / domains + 1;

    auto b = std::min(size, domain_size * domain);
    auto const e = std::min(size, b + domain_size);

    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) != -1; ++b, c = begin[b]);
    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) == -1; ++b, c = begin[b]);

    trie* n = &root;
    for (char c = begin[b]; ; ++b, c = begin[b]) {
        auto const idx = b >= size? -1 : idx(c);
        if (idx == -1) {
            if (n != &root) {
                n->count += 1;
                n = &root;
            }
            if (b >= size || b > e) break;
            else continue;
        }
        if (nullptr == n->children[idx].ptr) {
            auto next = bump++;
            n->children[idx].ptr = next;
        }
        n = n->children[idx].ptr;
    }
}
```

… this is an example of some text sub-domain …

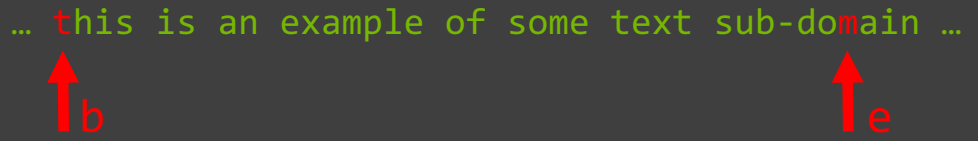
Trie append

```
void append(trie& root, trie*& bump, char const* begin, char const* end, unsigned domain, unsigned domains) {
    auto const size = end - begin;
    auto const domain_size = size / domains + 1;

    auto b = std::min(size, domain_size * domain);
    auto const e = std::min(size, b + domain_size);

    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) != -1; ++b, c = begin[b]);
    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) == -1; ++b, c = begin[b]);

    trie* n = &root;
    for (char c = begin[b]; ; ++b, c = begin[b]) {
        auto const idx = b >= size? -1 : idx(c);
        if (idx == -1) {
            if (n != &root) {
                n->count += 1;
                n = &root;
            }
            if (b >= size || b > e) break;
            else continue;
        }
        if (nullptr == n->children[idx].ptr) {
            auto next = bump++;
            n->children[idx].ptr = next;
        }
        n = n->children[idx].ptr;
    }
}
```



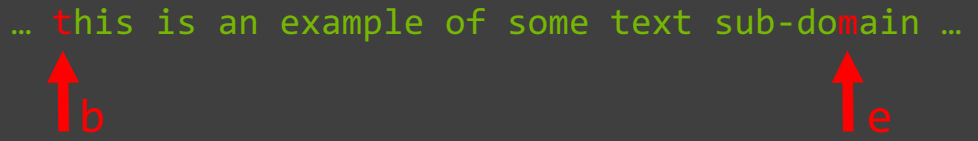
Trie append Prelude

```
void append(trie& root, trie*& bump, char const* begin, char const* end, unsigned domain, unsigned domains) {
    auto const size = end - begin;
    auto const domain_size = size / domains + 1;

    auto b = std::min(size, domain_size * domain);
    auto const e = std::min(size, b + domain_size);

    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) != -1; ++b, c = begin[b]);
    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) == -1; ++b, c = begin[b]);

    trie* n = &root;
    for (char c = begin[b]; ; ++b, c = begin[b]) {
        auto const idx = b >= size? -1 : idx(c);
        if (idx == -1) {
            if (n != &root) {
                n->count += 1;
                n = &root;
            }
            if (b >= size || b > e) break;
            else continue;
        }
        if (nullptr == n->children[idx].ptr) {
            auto next = bump++;
            n->children[idx].ptr = next;
        }
        n = n->children[idx].ptr;
    }
}
```



Trie append Body

```
void append(trie& root, trie*& bump, char const* begin, char const* end, unsigned domain, unsigned domains) {
    auto const size = end - begin;
    auto const domain_size = size / domains + 1;

    auto b = std::min(size, domain_size * domain);
    auto const e = std::min(size, b + domain_size);

    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) != -1; ++b, c = begin[b]);
    for (char c = begin[b]; b < size && b != e && c != 0 && idx_of(c) == -1; ++b, c = begin[b]);

    trie* n = &root;
    for (char c = begin[b]; ; ++b, c = begin[b]) {
        auto const idx = b >= size? -1 : idx(c);
        if (idx == -1) {
            if (n != &root) {
                n->count += 1;
                n = &root;
            }
            if (b >= size || b > e) break;
            else continue;
        }
        if (nullptr == n->children[idx].ptr) {
            auto next = bump++;
            n->children[idx].ptr = next;
        }
        n = n->children[idx].ptr;
    }
}
```



Trie append Body

Start at the root node of the tree…

…processing all characters…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If exceeded size of the domain or the character is not a letter (i.e. `idx(c) == -1`, e.g., due to whitespace, which separates words)…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If exceeded size of the domain or the character is not a letter (i.e. `idx(c) == -1`, e.g., due to whitespace, which separates words)…

If the node is not the root of the tree, then we just finished processing a word…

…so we increment the count, and go back to the root of the tree…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If exceeded size of the domain or the character is not a letter (i.e. `idx(c) == -1`, e.g., due to whitespace, which separates words)…

If the node is not the root of the tree, then we just finished processing a word…

…so we increment the count, and go back to the root of the tree…

…and we exit if this was the last word.

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If the current node does not have a child node at the character…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If the current node does not have a child node at the character…

…then:

1. We allocate a new node, and
2. set the child pointer.

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append Body

If the current node does not have a child node at the character…

…then:

1. We allocate a new node, and
2. set the child pointer.

…and then we traverse to the next node!

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append
Parallel version

To parallelize append...

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append
Parallel version

To parallelize append…

… incrementing the count needs to happen concurrently without data-races…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append
Parallel version

To parallelize append…

… incrementing the count needs to happen concurrently without data-races…

…need to prevent multiple threads from allocating the same node by “locking”…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append
Parallel version

To parallelize append…

… incrementing the count needs to happen concurrently without data-races…

…need to prevent multiple threads from allocating the same node by “locking”…

…need to load pointers concurrently…

```
trie* n = &root;
for (char c = begin[b]; ; ++b, c = begin[b]) {
    auto const idx = b >= size? -1 : idx(c);
    if (idx == -1) {
        if (n != &root) {
            n->count += 1;
            n = &root;
        }
        if (b >= size || b > e) break;
        else continue;
    }
    if (nullptr == n->children[idx].ptr) {
        auto next = bump++;
        n->children[idx].ptr = next;
    }
    n = n->children[idx].ptr;
}
```

Trie append
Double-checked locking

Parallelize the allocation…

```
if (nullptr == n->children[idx].ptr) {
  auto next = bump++;
  n->children[idx].ptr = next;
}
n = n->children[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

```
std::atomic<trie*>& bump;

if (nullptr == n->children[idx].ptr) {
  auto next = bump++;
  n->children[idx].ptr = next;
}
n = n->children[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

…extend trie reference with flag to allow locking while a thread is allocating…

```
std::atomic<trie*>& bump;

struct trie::ref {
    std::atomic<trie*> ptr;
    std::atomic_flag flag;
};

if (nullptr == n->children[idx].ptr) {
  auto next = bump++;
  n->children[idx].ptr = next;
}
n = n->children[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

…extend trie reference with flag to allow locking while a thread is allocating…

Double-checked locking:

1. If there is a child node, exit without lock

```
std::atomic<trie*>& bump;

struct trie::ref {
    std::atomic<trie*> ptr;
    std::atomic_flag flag;
};

if (nullptr == *n->children[idx].ptr) {
    if (!n->next[idx].flag.test_and_set()) {
        auto next = bump++;
        p->next[idx].ptr = next;
        n->next[idx].ptr.notify_all();
    } else
        p->next[idx].ptr.wait(nullptr);
}
n = *n->next[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

…extend trie reference with flag to allow locking while a thread is allocating…

Double-checked locking:

1. If there is a child node, exit without lock
2. Threads set the flag -> first thread false

```
std::atomic<trie*>& bump;

struct trie::ref {
    std::atomic<trie*> ptr;
    std::atomic_flag flag;
};

if (nullptr == *n->children[idx].ptr) {
    if (!n->next[idx].flag.test_and_set()) {
        auto next = bump++;
        p->next[idx].ptr = next;
        n->next[idx].ptr.notify_all();
    } else
        p->next[idx].ptr.wait(nullptr);
}
n = *n->next[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

…extend trie reference with flag to allow locking while a thread is allocating…

Double-checked locking:

1. If there is a child node, exit without lock
2. Threads set the flag -> first thread false
    1. First thread allocates node

```
std::atomic<trie*>& bump;

struct trie::ref {
    std::atomic<trie*> ptr;
    std::atomic_flag flag;
};

if (nullptr == *n->children[idx].ptr) {
    if (!n->next[idx].flag.test_and_set()) {
        auto next = bump++;
        p->next[idx].ptr = next;
        n->next[idx].ptr.notify_all();
    } else
        p->next[idx].ptr.wait(nullptr);
}
n = *n->next[idx].ptr;
```

Trie append
Double-checked locking

Parallelize allocation by making bump allocator atomic…

…extend trie reference with flag to allow locking while a thread is allocating…

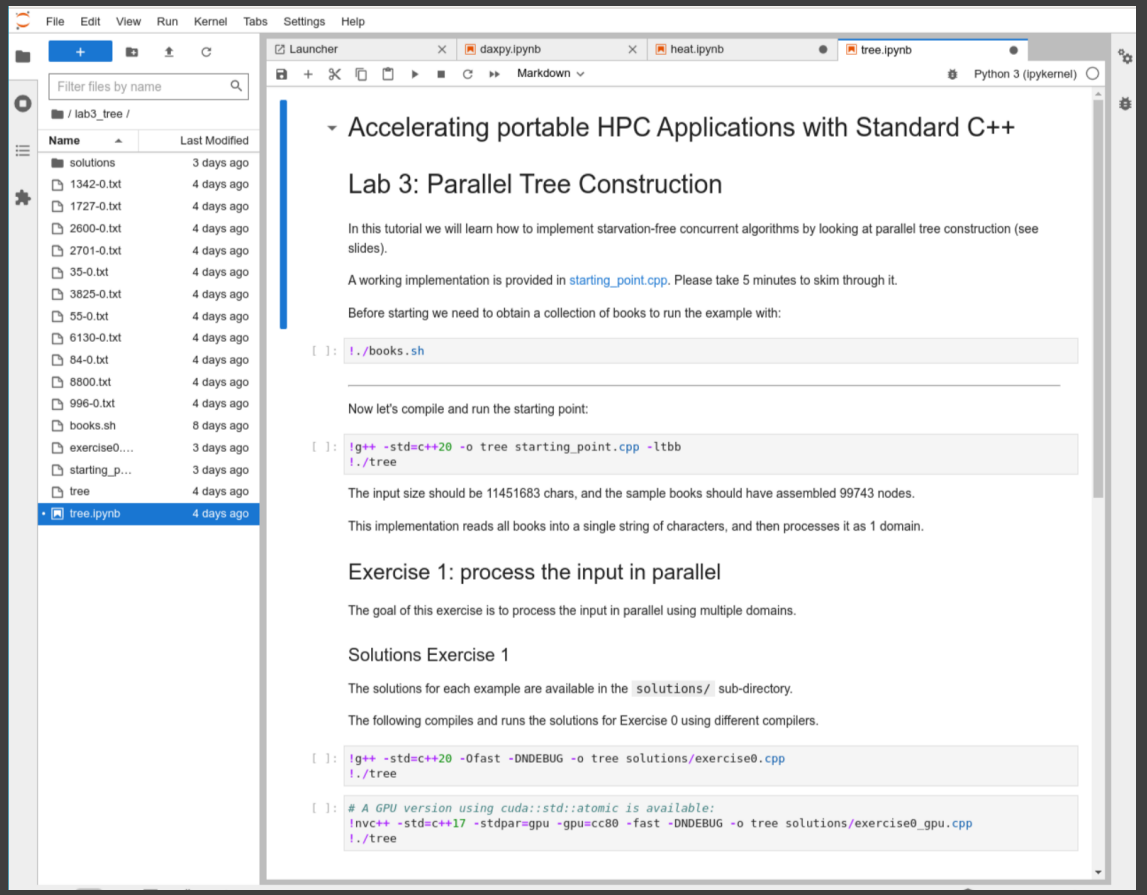
Double-checked locking:

1. If there is a child node, exit without lock
2. Threads set the flag -> first thread false
    1. First thread allocates node
    2. Rest waits on node being allocated

```
std::atomic<trie*>& bump;

struct trie::ref {
    std::atomic<trie*> ptr;
    std::atomic_flag flag;
};

if (nullptr == *n->children[idx].ptr) {
    if (!n->next[idx].flag.test_and_set()) {
        auto next = bump++;
        p->next[idx].ptr = next;
        n->next[idx].ptr.notify_all();
    } else
        p->next[idx].ptr.wait(nullptr);
}
n = *n->next[idx].ptr;
```

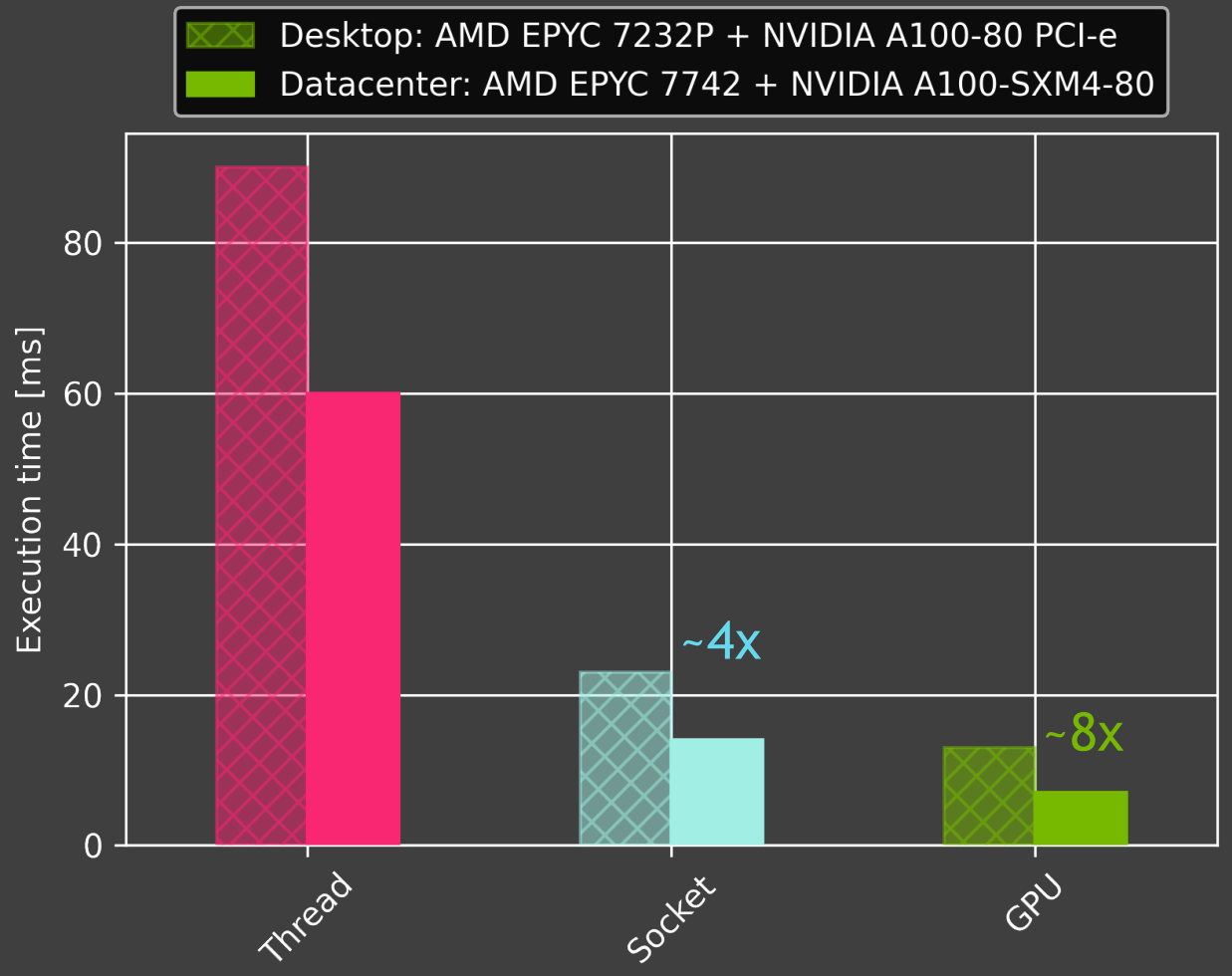
Lab 3: Parallel Tree Construction

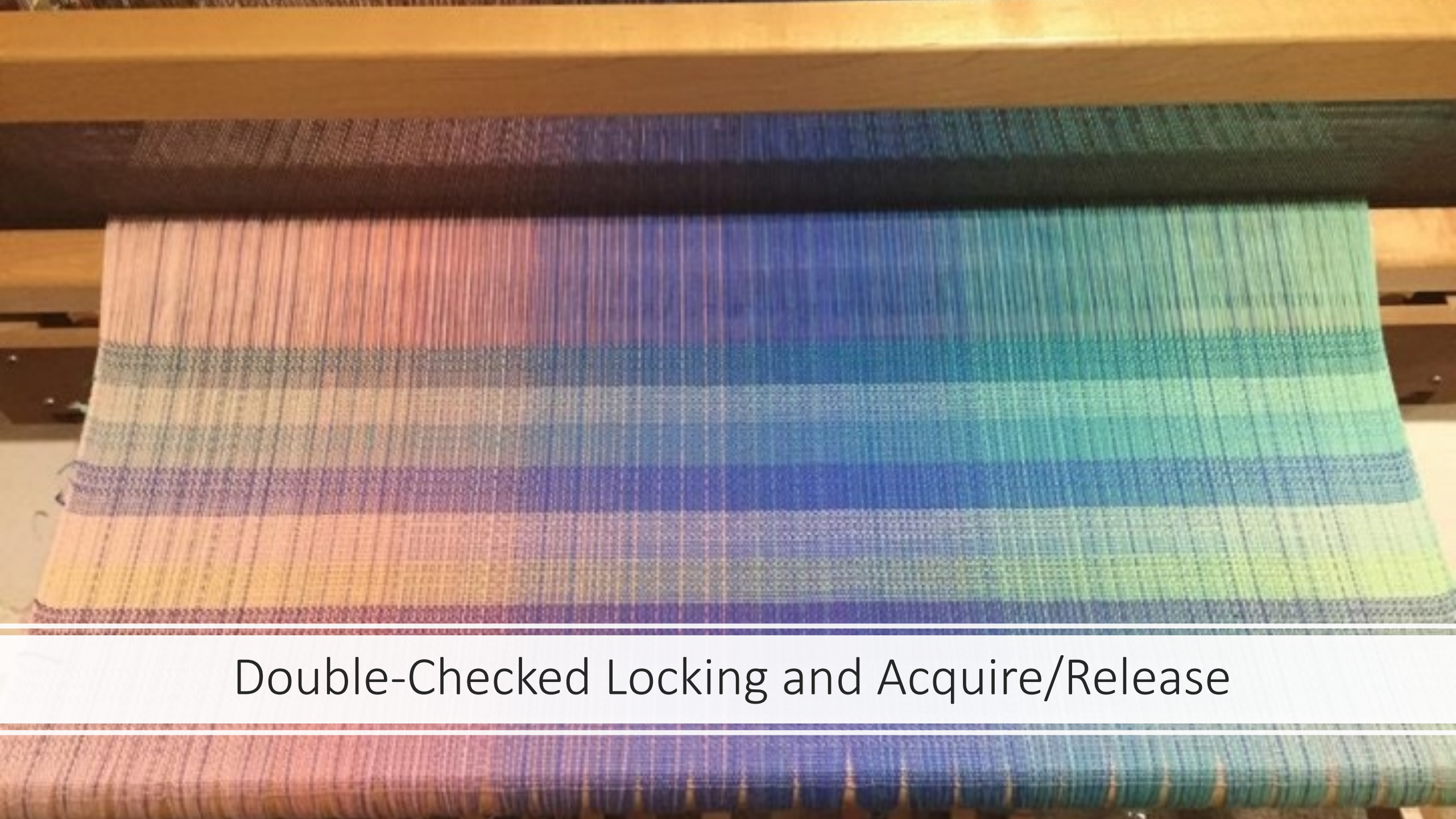
- **Exercise 1:** build the Trie in parallel using the concurrency primitives demonstrated above to create a critical section for allocating new nodes.



Lab 3: Parallel Tree Construction Solutions (DEMO)

Trie: data-structure to accelerate searches





Double-Checked Locking and Acquire/Release

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(int& flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Signals other threads that data is ready
    flag = 1;
}
```

Consumer Thread

```
int poll_then_read(int& flag, int& data) {
    // Wait until data ready
    while (flag != 1);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(int& flag, int& data, int value) {
    // Writes to some data
    data = 42;  // UB: data race
    // Signals other threads that data is ready
    flag = 1;   // UB: data race
}
```

Consumer Thread

```
int poll_then_read(int& flag, int& data) {
    // Wait until data ready
    while (flag != 1);  // UB: data race
    // Reads data
    return data;        // UB: data-race
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_ref<int> flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Signals other threads that data is ready
    flag.store(1, memory_order_relaxed);
}
```

Consumer Thread

```
int poll_then_read(std::atomic_ref<int> flag, int& data) {
    // Wait until data ready
    while (flag.load(memory_order_relaxed) != 1);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_ref<int> flag, int& data, int value) {
    // Writes to some data
    data = 42; // UB: data race
    // Signals other threads that data is ready
    flag.store(1, memory_order_relaxed);
}
```

Consumer Thread

```
int poll_then_read(std::atomic_ref<int> flag, int& data) {
    // Wait until data ready
    while (flag.load(memory_order_relaxed) != 1);
    // Reads data
    return data;  // UB: data race
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_ref<int> flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Make sure pending memory operations completed
    std::atomic_thread_fence(memory_order_release);
    // Signals other threads that data is ready
    flag.store(1, memory_order_relaxed);
}
```

Consumer Thread

```
int poll_then_read(std::atomic_ref<int> flag, int& data) {
    // Wait until data ready
    while (flag.load(memory_order_relaxed) != 1);
    // Invalidate cached memory operations
    std::atomic_thread_fence(memory_order_acquire);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
Acquire / Release semantics

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_ref<int> flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Flush pending memops and signal that data is ready
    flag.store(1, memory_order_release);
}
```

Consumer Thread

```
int poll_then_read(std::atomic_ref<int> flag, int& data) {
    // Wait until data ready and invalidate cached memops
    while (flag.load(memory_order_acquire) != 1);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing
std::atomic wait and notify

Global memory

```
int flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_ref<int> flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Flush pending memops and signal that data is ready
    flag.store(1, memory_order_release);
    // Unblock all threads waiting
    flag.notify_all();
}
```

Consumer Thread

```
int poll_then_read(std::atomic_ref<int> flag, int& data) {
    // Wait until data ready and invalidate cached memops
    flag.wait(0, memory_order_acquire);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Message Passing std::atomic_flag

Global memory

```
std::atomic_flag flag = 0;
int data = 0;
```

Producer Thread

```
void write_then_signal(std::atomic_flag& flag, int& data, int value) {
    // Writes to some data
    data = 42;
    // Flush pending memops and signal that data is ready
    flag.test_and_set(memory_order_release);
    // Unblock all threads waiting
    flag.notify_all();
}
```

Consumer Thread

```
int poll_then_read(std::atomic_flag& flag, int& data) {
    // Wait until data ready and invalidate cached memops
    flag.wait(memory_order_acquire);
    // Reads data
    return data;
}
```

```
assert(flag == 0 || data == 42);
```

Concurrent initialization
First thread to arrive initializes data

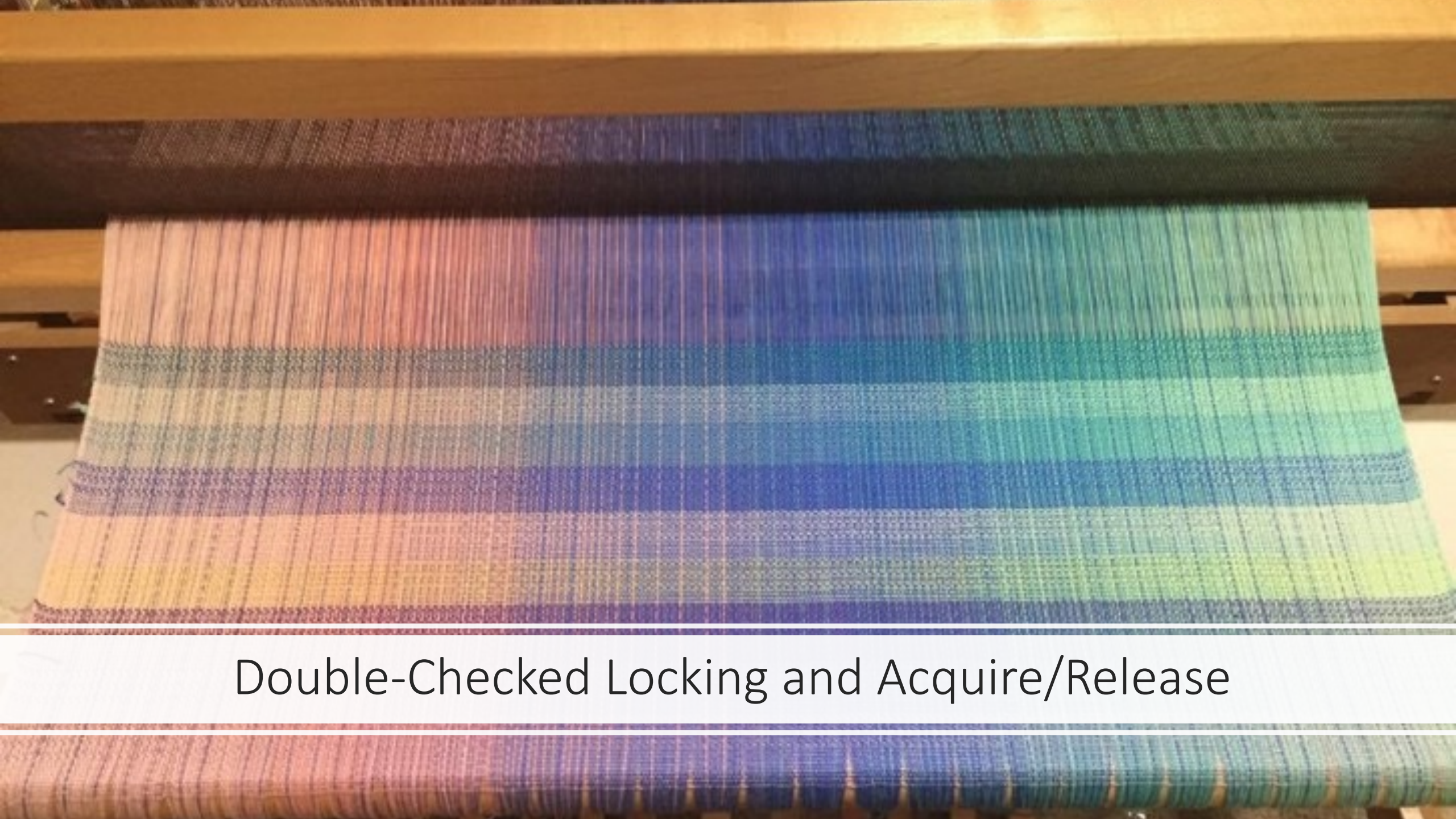
Global memory

```
std::atomic_flag flag = 0;
int* data = 0;
```

Producer and Consumer Threads

```
int* initialize(std::atomic_flag& flag, std::atomic_ref<int*> data, int value) {
    if (false == flag.test_and_set()) {
       // Only first thread sees the initial false value of flag
       data.store(new int(42), memory_order_release);
       data.notify_all();
    } else {
       // All other threads see true and wait for data to be initialized
       data.wait(0, memory_order_acquire);
    }
    return data.load(memory_order_relaxed);
}
```

```
assert(flag == 0 || data == 42);
```



Double-Checked Locking and Acquire/Release

The end

The end