GPU Acceleration with the C++ Standard Library

Intended Audience

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

Prerequisites: experience with C++11 (lambdas, STL algorithms)

GPU Acceleration with the C++ Standard Library

C++ Prerequisites

Fundamentals of ISO C++ Parallelism

Indexing, Ranges & Views

Interactive Materials

C++ Prerequisites

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…

```cpp
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);
```

…is equivalent to…

```cpp
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)

```cpp
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

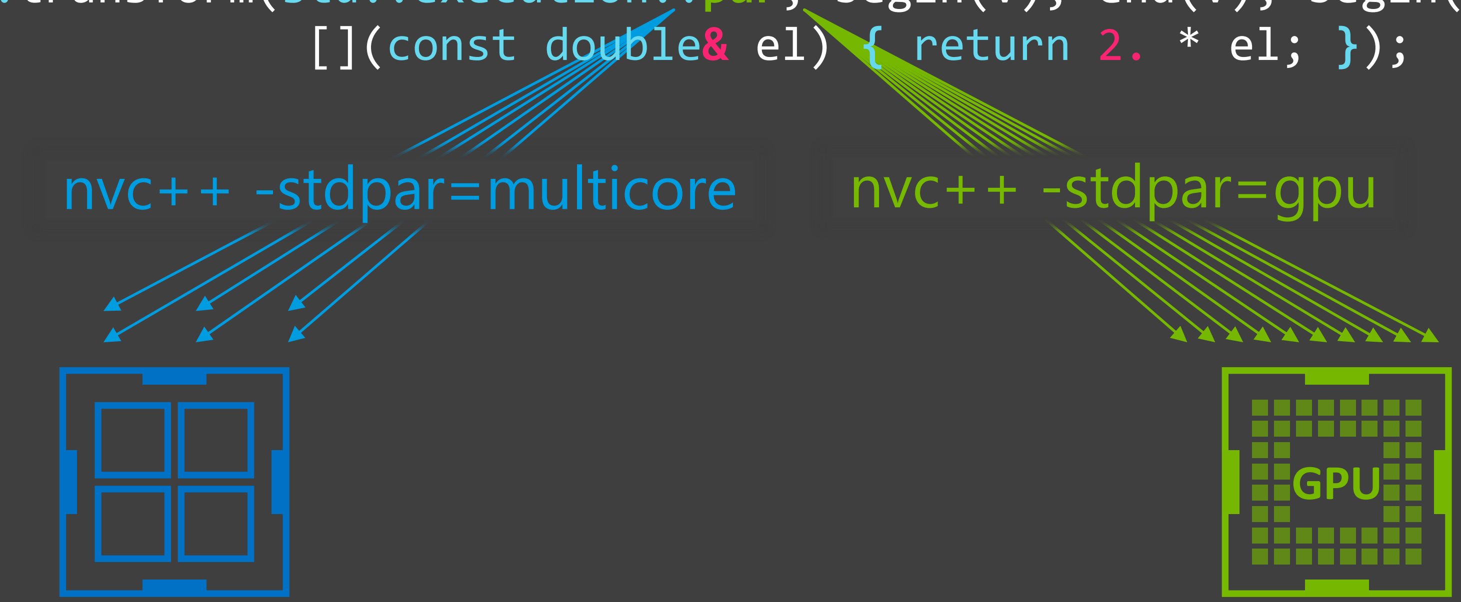
```
std::vector<double> v = {1, 2, 3, 4}, w(4);
std::transform(std::execution::par, begin(v), end(v), begin(w),
                [](const double& el) { return 2. * el; });
```

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; });
```



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(v), end(v), 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 unified (managed) memory model.

ISO C++ parallel algorithms
Accelerator support limitation

Stack variable “a” and global variable “b” are 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 and globals only, not for heap data (the vector).

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

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_scan

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...

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

Parallel versions of the algorithms do not!

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, ….
```

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

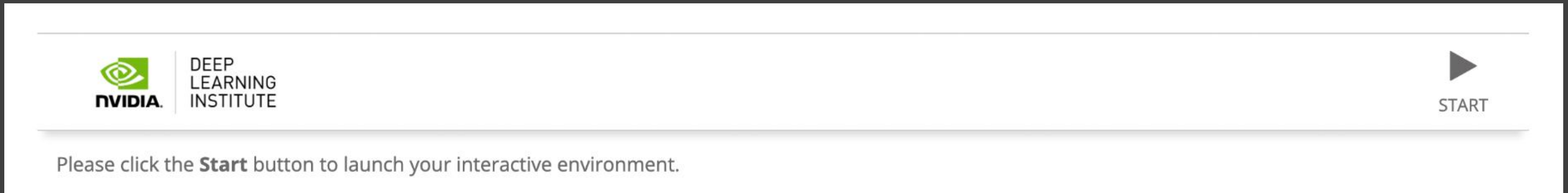
```
std::vector<int> w{4, 5, 6, 7};
for (auto [x, y] : w | range::views::enumerate)
    cout << "(" << x << "," << y << "), ";

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

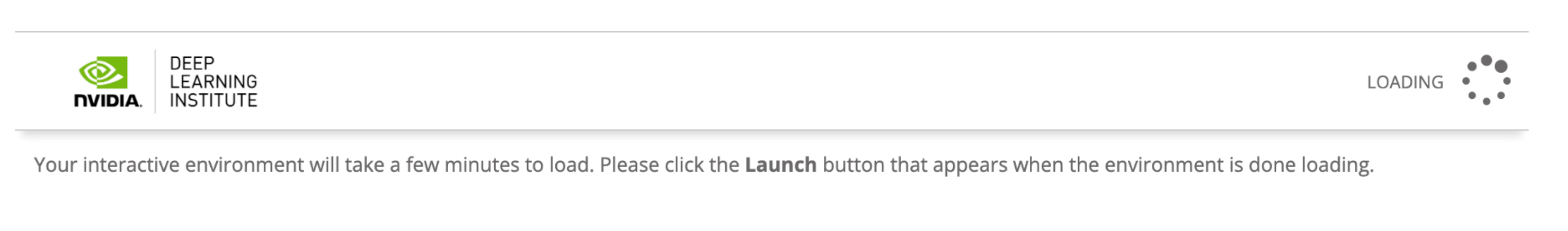
References

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

Interactive Materials

Launch the Interactive Environment





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...

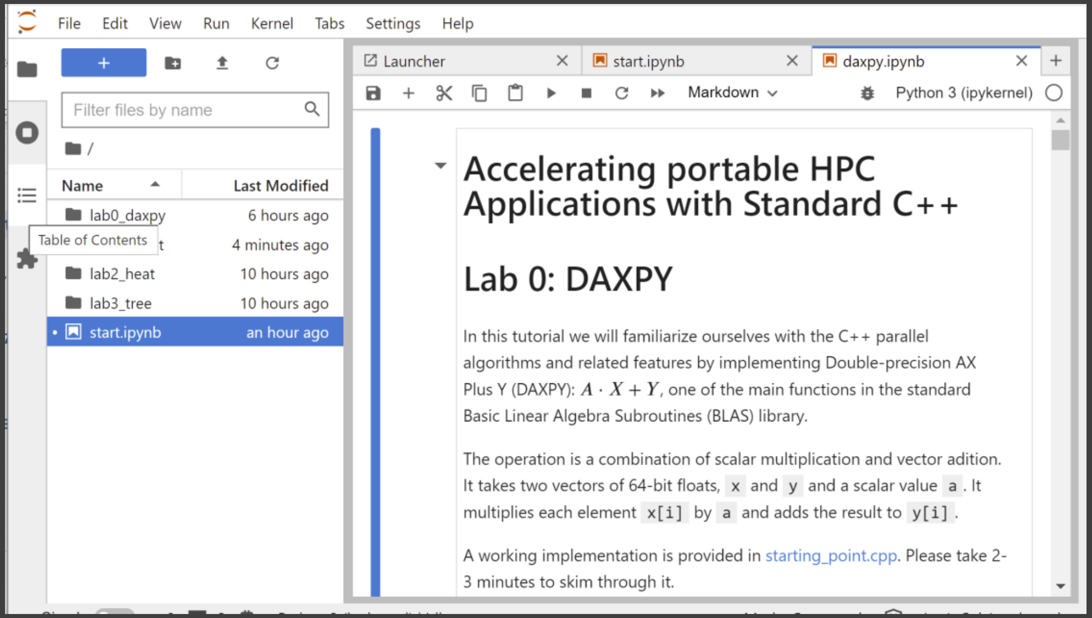
```
for (int i = 0; i < N; ++i) {
    y[i] += a * x[i];
}
```

BLAS DAXPY
3 Exercises

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

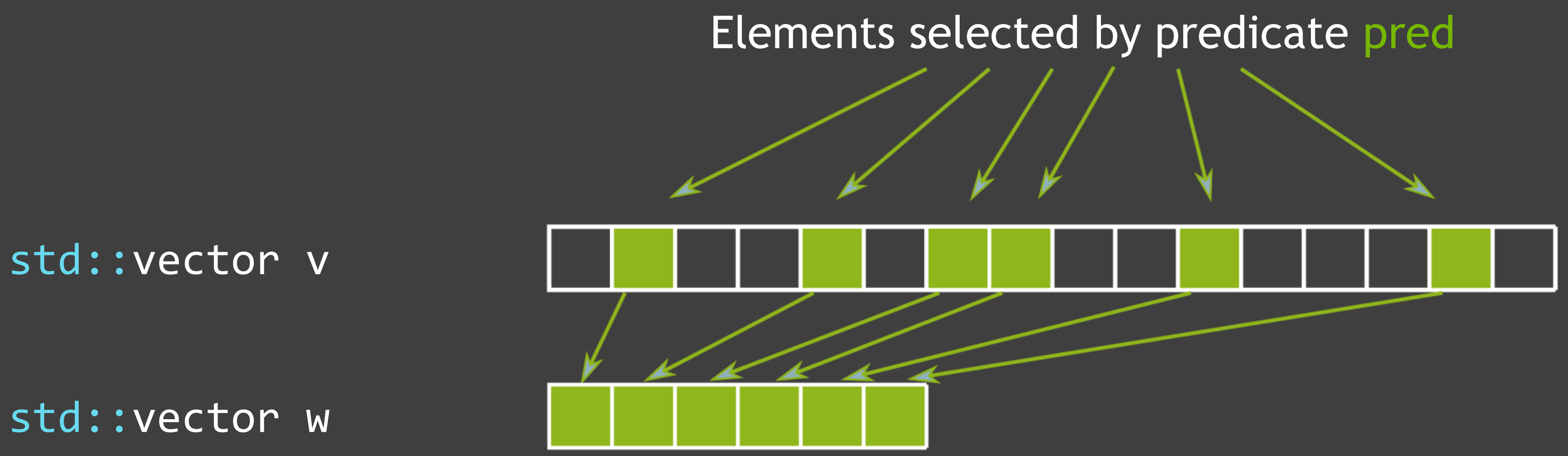
- **Exercise 2:** rewrite the for-loop implementation of the “initialization” kernel using sequential STL algorithms with any of the indexing approaches discussed in the tutorial

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



Extra Credit: Select

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

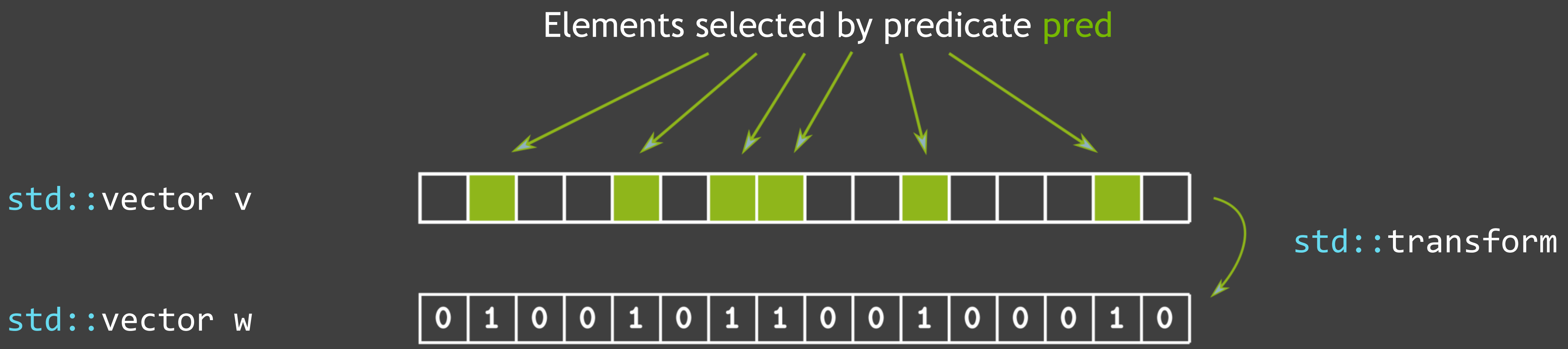


- Sequentially no problem, but how to write a parallel algorithm ?

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

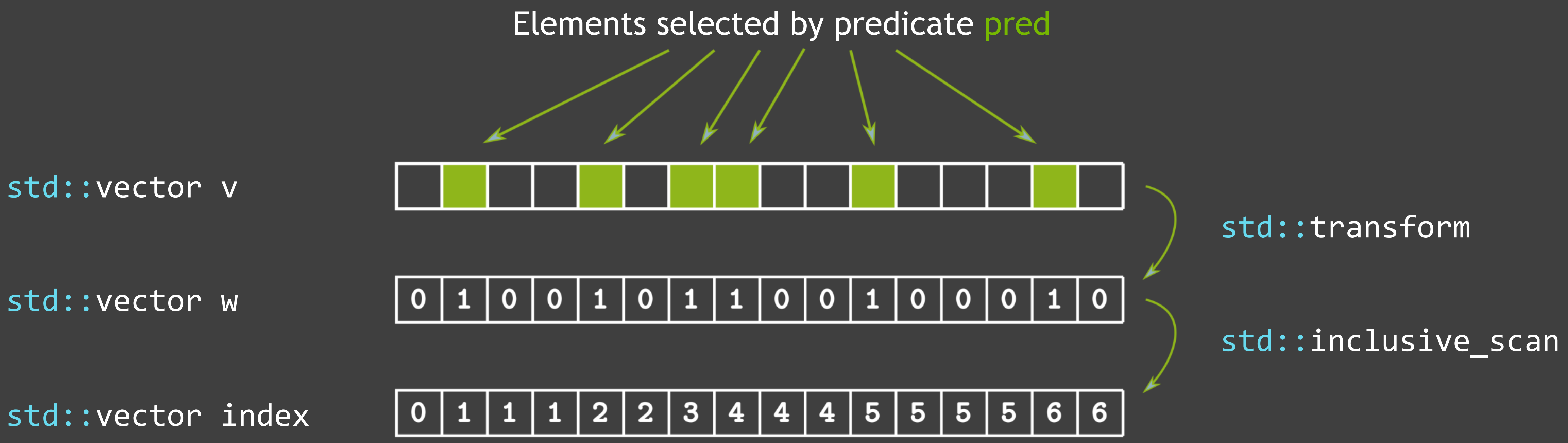
Extra Credit: Select

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



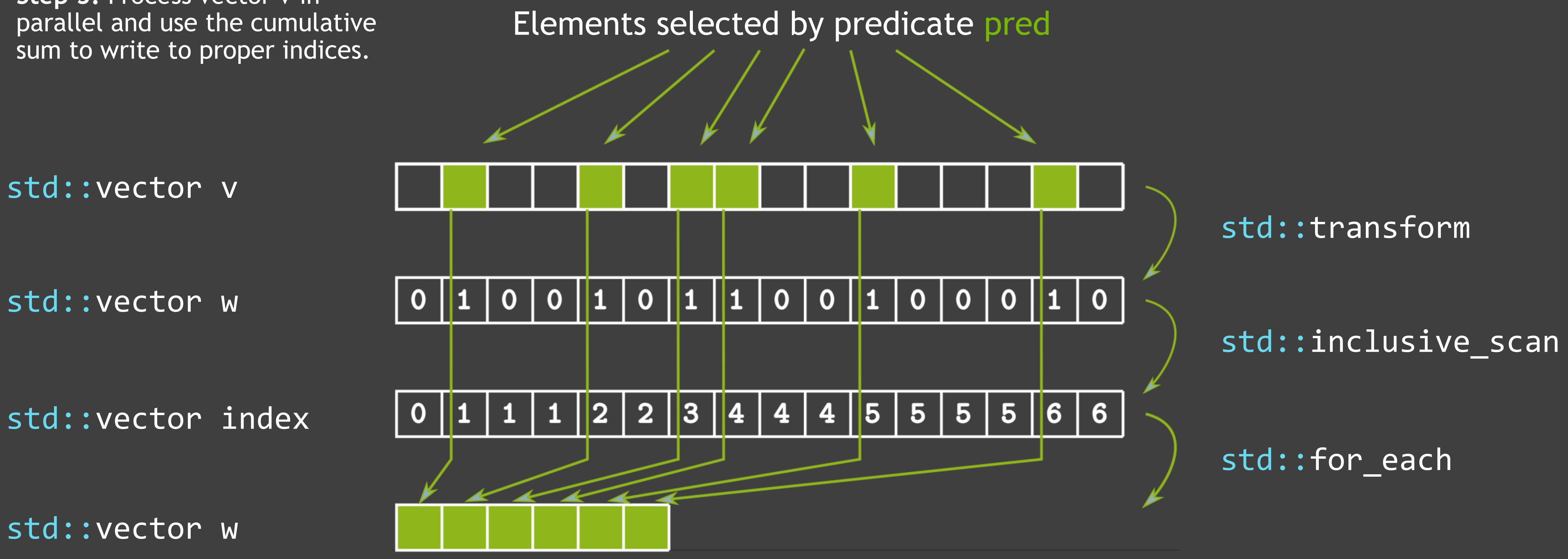
Extra Credit: Select

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



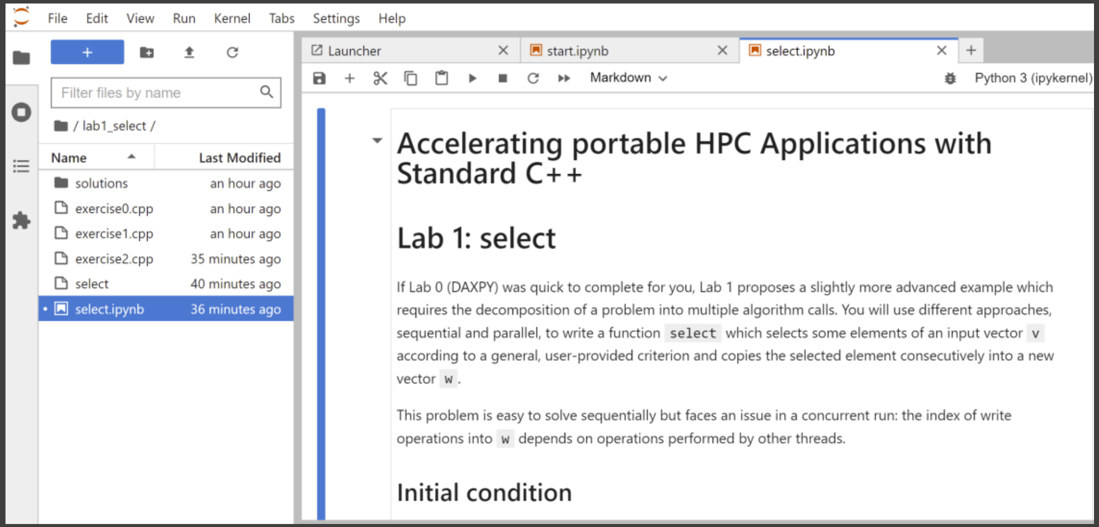
Extra Credit: Select

- **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.

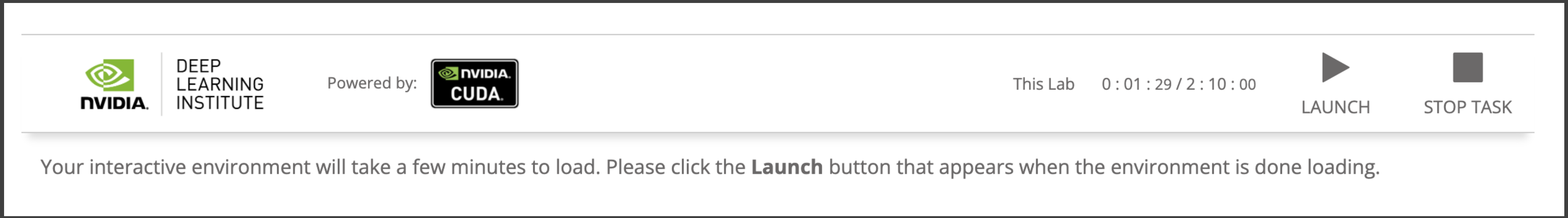


Extra Credit: Select
3 Exercises

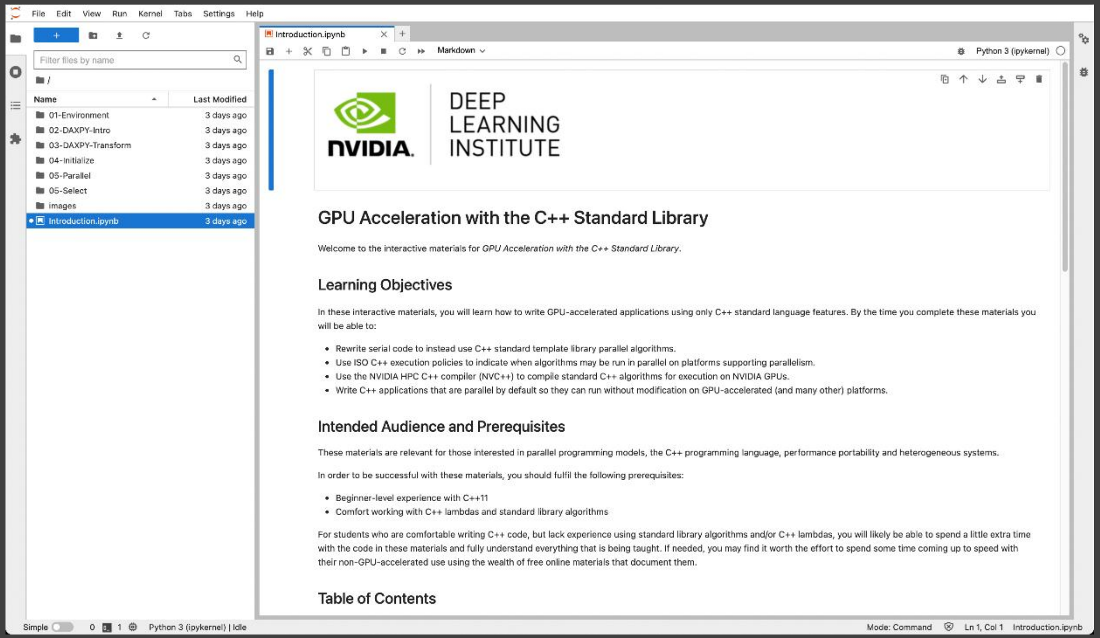
- Exercise 1: write a sequential version of the function select that uses the algorithm std::copy_if and a back inserter for vector w.
- **Exercise 2:** write a parallel version of the function select that works with two temporary vectors v_sel and index.
- **Exercise 3:** reduce the number of steps from 3 to 2 and avoid the creation of v_sel using the algorithm transform_inclusive_scan.



The Interactive Materials



The Interactive Materials



File Edit View Run Kernel Tabs Settings Help

Name	Last Modified
01-Environment	3 days ago
02-DAXPY-Intro	3 days ago
03-DAXPY-Transform	3 days ago
04-Initialize	3 days ago
05-Parallel	3 days ago
05-Select	3 days ago
images	3 days ago
Introduction.ipynb	3 days ago

DEEP LEARNING INSTITUTE

GPU Acceleration with the C++ Standard Library

Welcome to the interactive materials for *GPU Acceleration with the C++ Standard Library*.

Learning Objectives

In these interactive materials, you will learn how to write GPU-accelerated applications using only C++ standard language features. By the time you complete these materials you will be able to:

- Rewrite serial code to instead use C++ standard template library parallel algorithms.
- Use ISO C++ execution policies to indicate when algorithms may be run in parallel on platforms supporting parallelism.
- Use the NVIDIA HPC C++ compiler (NVC++) to compile standard C++ algorithms for execution on NVIDIA GPUs.
- Write C++ applications that are parallel by default so they can run without modification on GPU-accelerated (and many other) platforms.

Intended Audience and Prerequisites

These materials are relevant for those interested in parallel programming models, the C++ programming language, performance portability and heterogeneous systems.

In order to be successful with these materials, you should fulfil the following prerequisites:

- Beginner-level experience with C++11
- Comfort working with C++ lambdas and standard library algorithms

For students who are comfortable writing C++ code, but lack experience using standard library algorithms and/or C++ lambdas, you will likely be able to spend a little extra time with the code in these materials and fully understand everything that is being taught. If needed, you may find it worth the effort to spend some time coming up to speed with their non-GPU-accelerated use using the wealth of free online materials that document them.

Table of Contents

Simple 0 1 Python 3 (ipykernel) | Idle Mode: Command Ln 1, Col 1 Introduction.ipynb

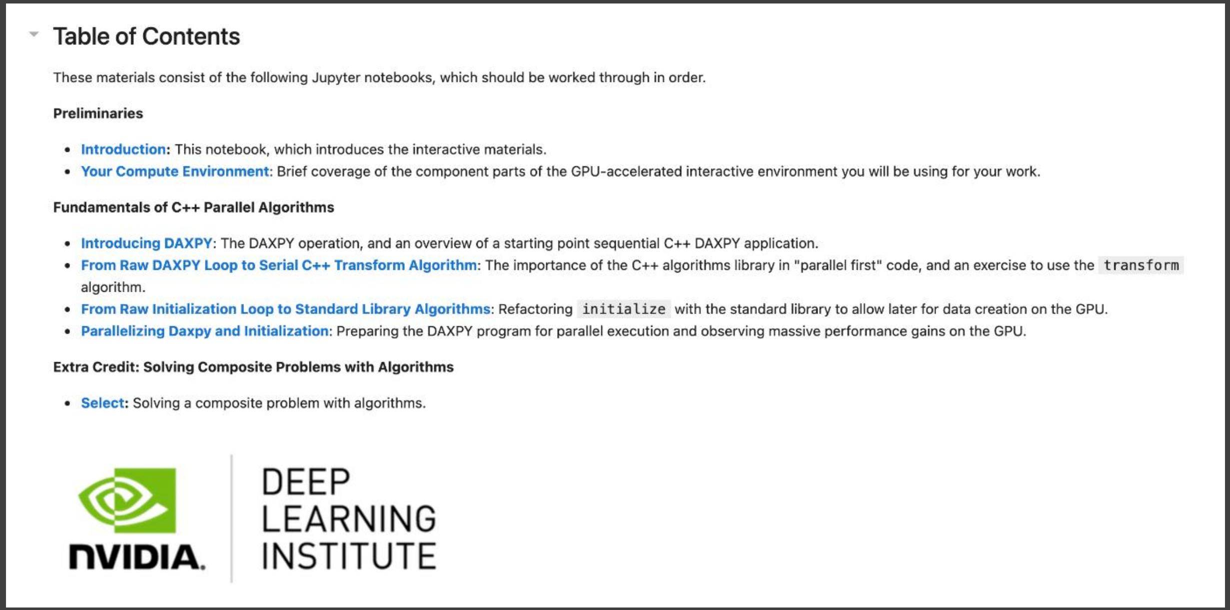


Table of Contents

These materials consist of the following Jupyter notebooks, which should be worked through in order.

Preliminaries

- **Introduction:** This notebook, which introduces the interactive materials.
- **Your Compute Environment**: Brief coverage of the component parts of the GPU-accelerated interactive environment you will be using for your work.

Fundamentals of C++ Parallel Algorithms

- **Introducing DAXPY**: The DAXPY operation, and an overview of a starting point sequential C++ DAXPY application.
- **From Raw DAXPY Loop to Serial C++ Transform Algorithm**: The importance of the C++ algorithms library in "parallel first" code, and an exercise to use the `transform` algorithm.
- **From Raw Initialization Loop to Standard Library Algorithms**: Refactoring `initialize` with the standard library to allow later for data creation on the GPU.
- **Parallelizing Daxpy and Initialization**: Preparing the DAXPY program for parallel execution and observing massive performance gains on the GPU.

Extra Credit: Solving Composite Problems with Algorithms

- **Select:** Solving a composite problem with algorithms.

DEEP LEARNING INSTITUTE

GPU Acceleration with the C++ Standard Library