Getting Started with Accelerated Computing in Modern CUDA C++

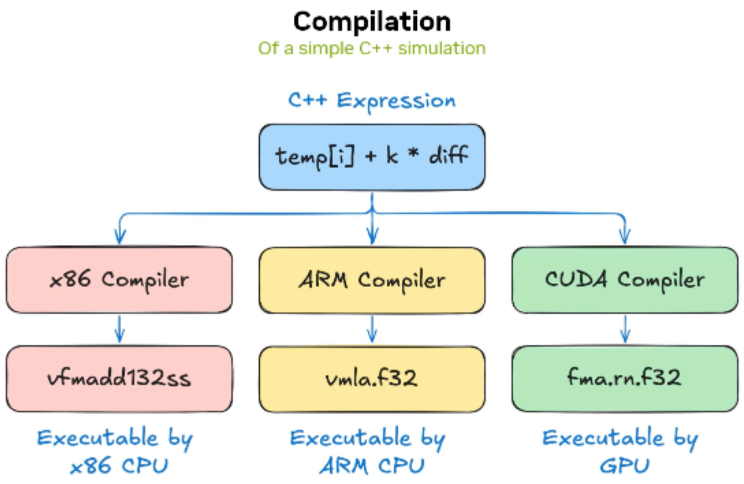
Aashay Kulkarni

January, 2026

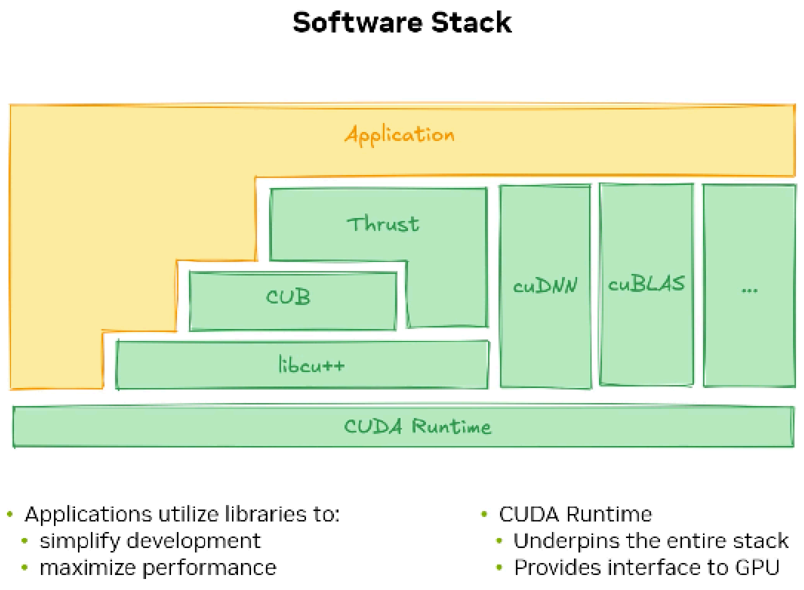
Contents

1. Introduction

- GPU has higher latency than the CPU, however, it has higher bandwidth. (Think of Bus vs Car)
- For problems with high amounts of data (i.e. for really large **N**) on which simultaneous computations can be performed (known as Parallel Computing) GPU Programming is better.
- C++ Code is compiled into architecture specific instructions (x86, ARM, CUDA) by a compiler.



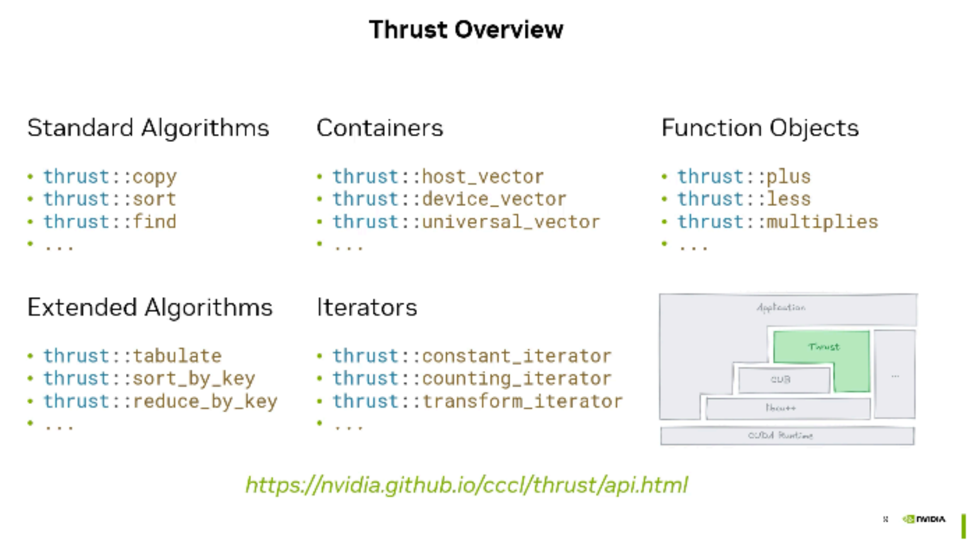
NVCC (NVIDIA CUDA Compiler Driver) turns C++ Code into GPU instructions. However, the GPU specific code must be stated explicitly. Execution Spaces are divided into host (CPU) and device (GPU). By default, code is run on the host side. By invoking special functions understood by the GPU compiler known as *kernels*, one can switch the execution space to a GPU.



CUDA provides lots of GPU accelerated libraries such as Thrust, cuBLAS, cuB, libcu++, cuDNN.

1.1. Thrust Library

Thrust simplifies GPU Programming by offering pre-built components for simple tasks.



Thrust Overview

Standard Algorithms

- thrust::copy
- thrust::sort
- thrust::find
- ...

Containers

- thrust::host_vector
- thrust::device_vector
- thrust::universal_vector
- ...

Function Objects

- thrust::plus
- thrust::less
- thrust::multiplies
- ...

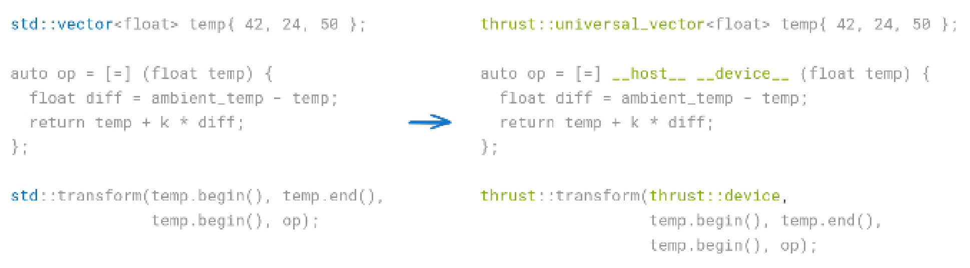
Extended Algorithms

- thrust::tabulate
- thrust::sort_by_key
- thrust::reduce_by_key
- ...

Iterators

- thrust::constant_iterator
- thrust::counting_iterator
- thrust::transform_iterator
- ...

https://nvidia.github.io/cccl/thrust/api.html



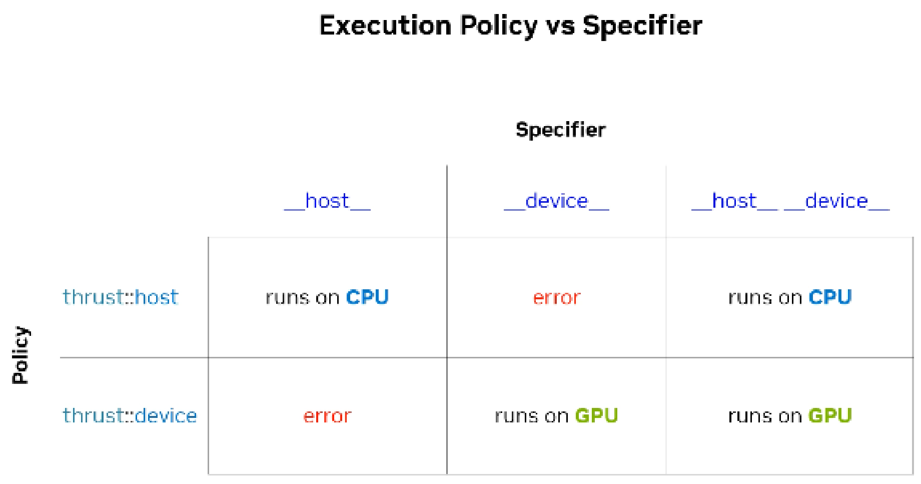
```
std::vector<float> temp{ 42, 24, 50 };

auto op = [=] (float temp) {
  float diff = ambient_temp - temp;
  return temp + k * diff;
};

std::transform(temp.begin(), temp.end(),
               temp.begin(), op);
```

```
thrust::universal_vector<float> temp{ 42, 24, 50 };

auto op = [=] __host__ __device__ (float temp) {
  float diff = ambient_temp - temp;
  return temp + k * diff;
};

thrust::transform(thrust::device,
                  temp.begin(), temp.end(),
                  temp.begin(), op);
```

1.2. Thrust Execution Policy

The first parameter to `thrust::transform()` tells Thrust where to run the algorithm. thrust::host executes algorithms on the CPU, while thrust::device executes algorithms on the GPU.

- Execution Policy vs Execution Specifier:
  - Execution space specifier (`__host__`, `__device__`) indicates where the code can run and it works at compile time.
  - Execution Policy: It works at runtime and indicates where the code will run. It doesn't automatically compile code for that location.

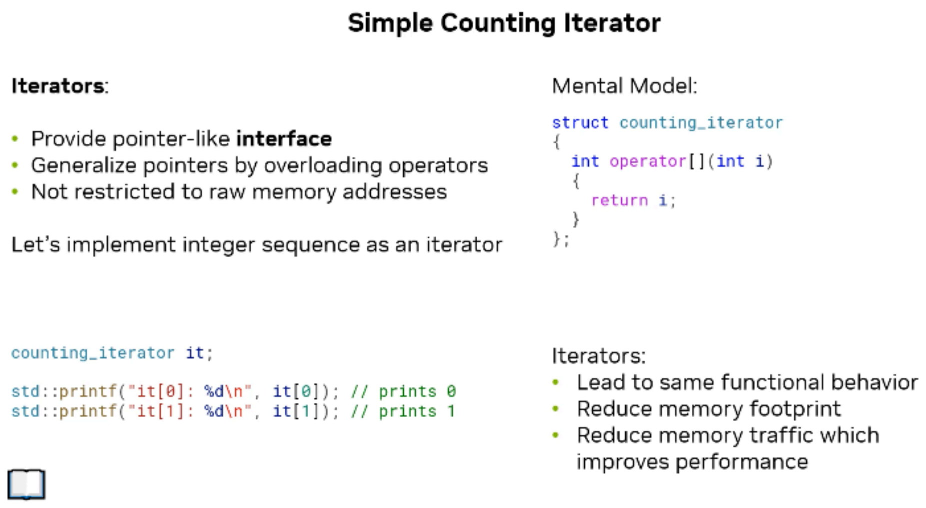


Execution Policy vs Specifier

Policy \ Specifier	`__host__`	`__device__`	`__host__ __device__`
thrust::host	runs on **CPU**	error	runs on **CPU**
thrust::device	error	runs on **GPU**	runs on **GPU**

1.3. Extending Standard algorithms

- Using Iterators will help reduce the # of memory accesses.
- Examples include counting iterators, zip iterators, transform iterators.
- Transform iterator - applies a function before returning a value. Zip Iterator - references 2 sequences at once.
- Using pointers means you will access memory every single time subscript operator is used. Using a counting iterator will just return the index at which the relevant data is stored.



Simple Counting Iterator

Iterators:

- Provide pointer-like **interface**
- Generalize pointers by overloading operators
- Not restricted to raw memory addresses

Let's implement integer sequence as an iterator

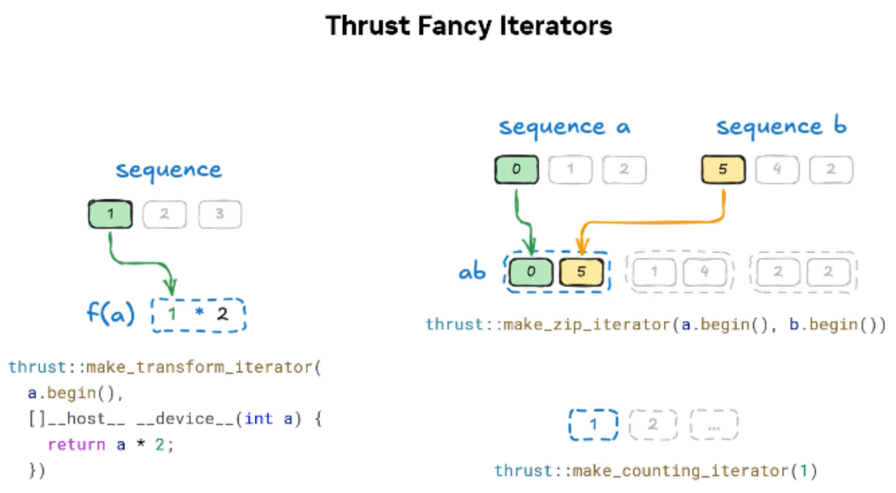
Mental Model:

```
struct counting_iterator
{
  int operator[](int i)
  {
    return i;
  }
};
```

```
counting_iterator it;

std::printf("it[0]: %d\n", it[0]); // prints 0
std::printf("it[1]: %d\n", it[1]); // prints 1
```

Iterators:
- Lead to same functional behavior
- Reduce memory footprint
- Reduce memory traffic which improves performance

Thrust has inbuilt operators which we can use to perform reduction algorithms, speeding up the execution of the code.

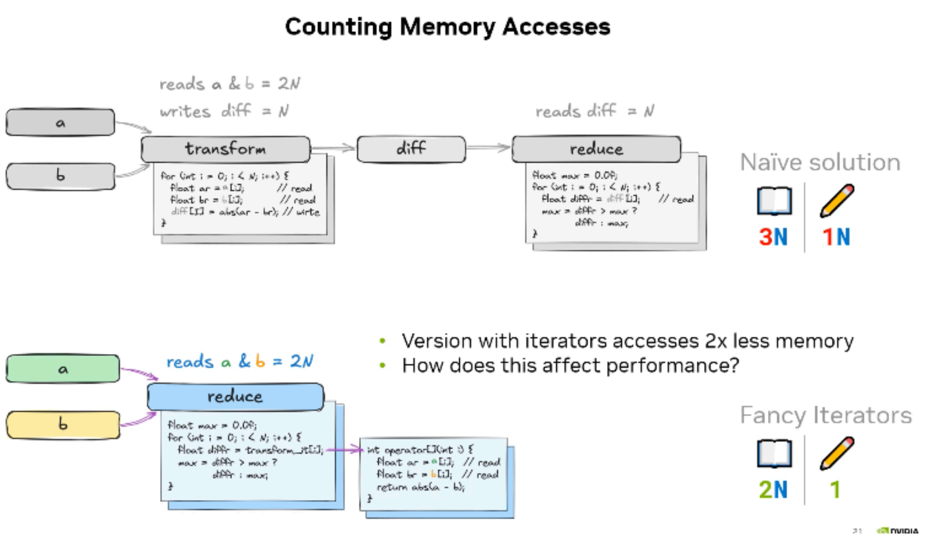


Thrust Fancy Iterators

```
thrust::make_transform_iterator(
  a.begin(),
  []__host__ __device__(int a) {
    return a * 2;
  })
```

```
thrust::make_zip_iterator(a.begin(), b.begin())
```

```
thrust::make_counting_iterator(1)
```



Counting Memory Accesses

```
for (int i = 0; i < N; i++) {
  float ar = a[i];           // read
  float br = b[i];           // read
  diff[i] = abs(ar - br); // write
}
```

```
float max = 0.0f;
for (int i = 0; i < N; i++) {
  float diffr = diff[i];     // read
  max = diffr > max ?
          diffr : max;
}
```

Naïve solution

3N 1N

- Version with iterators accesses 2x less memory
- How does this affect performance?

```
float max = 0.0f;
for (int i = 0; i < N; i++) {
  float diffr = transform_it[i];
  max = diffr > max ?
          diffr : max;
}
```

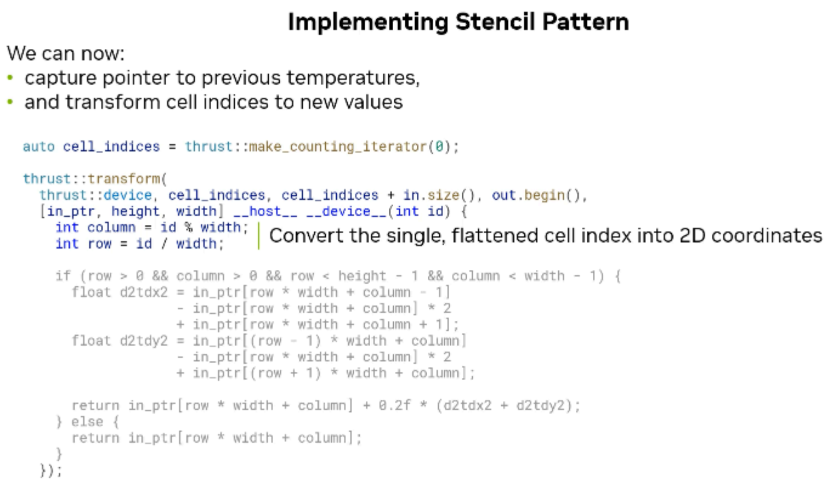
```
int operator[](int i) {
  float ar = a[i];  // read
  float br = b[i];  // read
  return abs(a - b);
}
```

Fancy Iterators

2N 1

1.4. Vocabulary Types

- In the below picture, the stencil is the id which is used to find the 2D coordinates of the data.
- Using Thrust containers, using data() will return a typed iterator while std will return a raw pointer.
- Thrust's typed iterator can be converted to a raw pointer using thrust::raw_pointer_cast



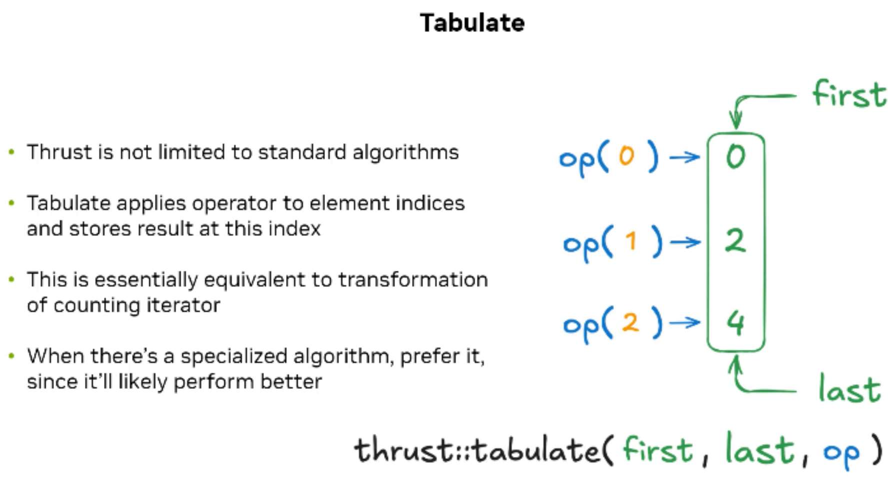
Implementing Stencil Pattern

We can now:
- capture pointer to previous temperatures,
- and transform cell indices to new values

```
auto cell_indices = thrust::make_counting_iterator(0);

thrust::transform(
  thrust::device, cell_indices, cell_indices + in.size(), out.begin(),
  [in_ptr, height, width] __host__ __device__(int id) {
    int column = id % width;
    int row = id / width;

    if (row > 0 && column > 0 && row < height - 1 && column < width - 1) {
      float d2tdx2 = in_ptr[row * width + column - 1]
                   - in_ptr[row * width + column] * 2
                   + in_ptr[row * width + column + 1];
      float d2tdy2 = in_ptr[(row - 1) * width + column]
                   - in_ptr[row * width + column] * 2
                   + in_ptr[(row + 1) * width + column];

      return in_ptr[row * width + column] + 0.2f * (d2tdx2 + d2tdy2);
    } else {
      return in_ptr[row * width + column];
    }
  });
```

Convert the single, flattened cell index into 2D coordinates

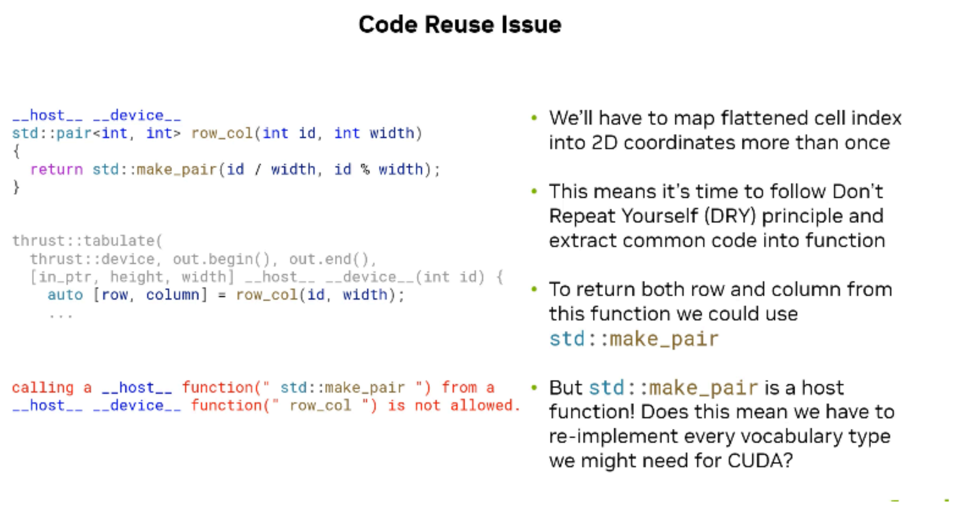
- Thrust Tabulate can be used to apply a common algorithm which can help parallelize the operation on the GPU.



Tabulate

- Thrust is not limited to standard algorithms
- Tabulate applies operator to element indices and stores result at this index
- This is essentially equivalent to transformation of counting iterator
- When there's a specialized algorithm, prefer it, since it'll likely perform better

- Below, we call std::make_pair, a host function, in a host, device function. This is not allowed and will not compile.



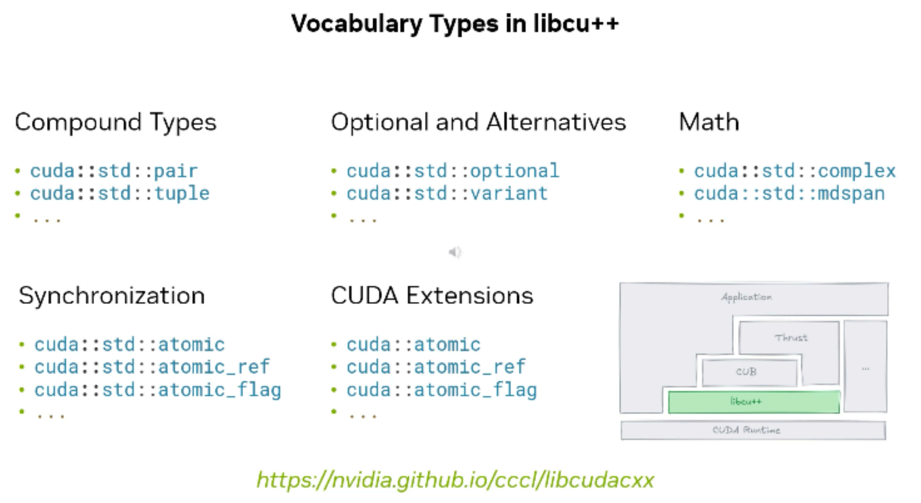
Code Reuse Issue

```
__host__ __device__
std::pair<int, int> row_col(int id, int width)
{
  return std::make_pair(id / width, id % width);
}


thrust::tabulate(
  thrust::device, out.begin(), out.end(),
  [in_ptr, height, width] __host__ __device__(int id) {
    auto [row, column] = row_col(id, width);
    ...


calling a __host__ function(" std::make_pair ") from a
__host__ __device__ function(" row_col ") is not allowed.
```

- We'll have to map flattened cell index into 2D coordinates more than once
- This means it's time to follow Don't Repeat Yourself (DRY) principle and extract common code into function
- To return both row and column from this function we could use `std::make_pair`
- But `std::make_pair` is a host function! Does this mean we have to re-implement every vocabulary type we might need for CUDA?

1.4.1. libcu++

- To fix this, we use the libcu++ library.
- This contains common std library types such as cuda::std::pair instead of std::pair.



Vocabulary Types in libcu++

Compound Types
- `cuda::std::pair`
- `cuda::std::tuple`
- ...

Optional and Alternatives
- `cuda::std::optional`
- `cuda::std::variant`
- ...

Math
- `cuda::std::complex`
- `cuda::std::mdspan`
- ...

Synchronization
- `cuda::std::atomic`
- `cuda::std::atomic_ref`
- `cuda::std::atomic_flag`
- ...

CUDA Extensions
- `cuda::atomic`
- `cuda::atomic_ref`
- `cuda::atomic_flag`
- ...

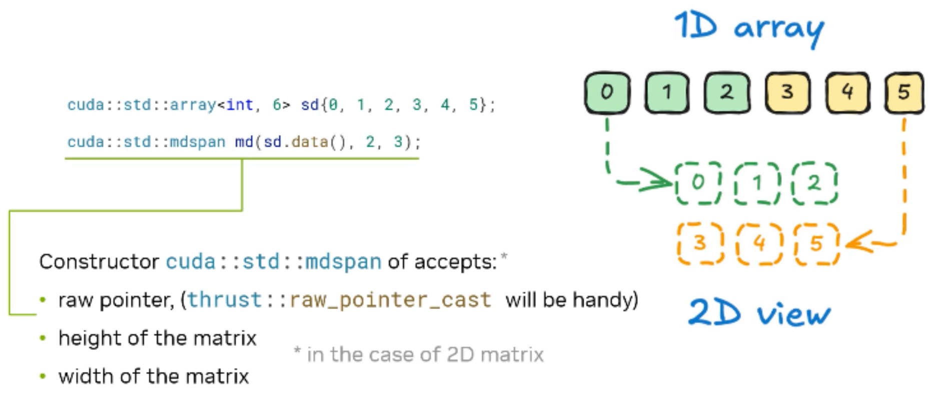
Application
Thrust
CUB
libcu++
...
CUDA Runtime

https://nvidia.github.io/cccl/libcudacxx

1.4.2. mdspan

- disadvantages of manual linearization become more apparent as number of dimensions increases.
- cuda::std::mdspan is a non owning multidimensional view of sequence.
- mdspan is just a 2D view of the below array, cuda::std::array is the raw data structure which stores the data.

mdspan



```
cuda::std::array<int, 6> sd{0, 1, 2, 3, 4, 5};

cuda::std::mdspan md(sd.data(), 2, 3);
```

Constructor `cuda::std::mdspan` of accepts:*

- raw pointer, (`thrust::raw_pointer_cast` will be handy)
- height of the matrix
- width of the matrix

* in the case of 2D matrix

- The operator() can be used to access the underlying sequence: md(0,0) = 0, md(1, 2) = 5
- size() return the total number of elments in the view
- extent(r) returns extent at rank r, md.extent(0) = 2, md.extent(1) = 3