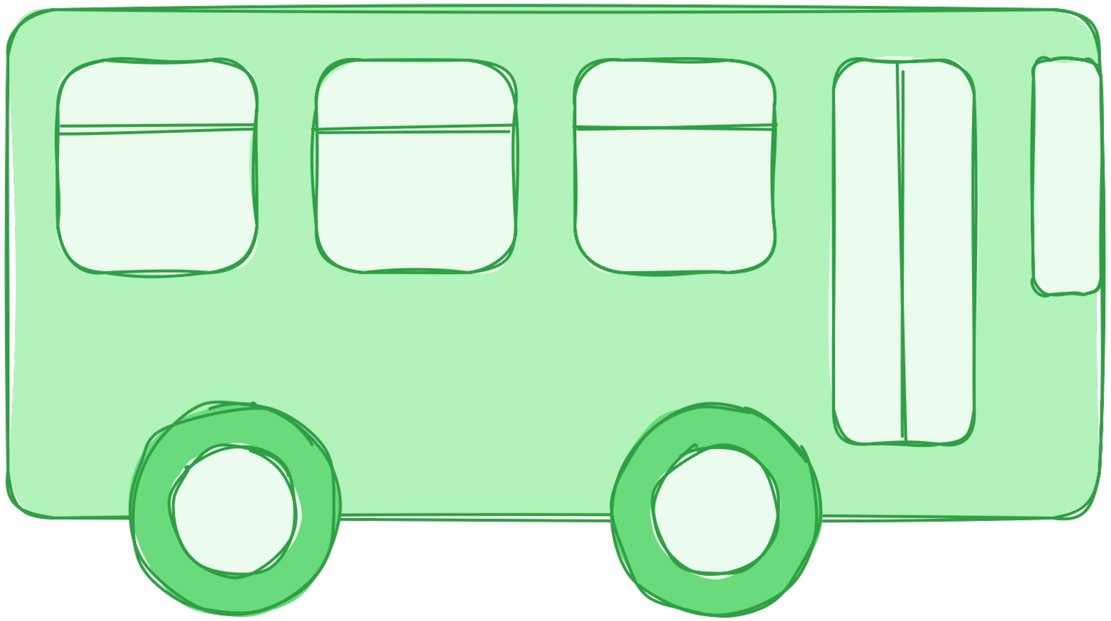
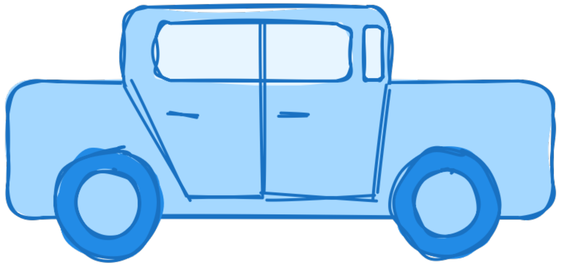
Fundamentals of Accelerated Computing with Modern CUDA C++

GPU Architecture at a Glance



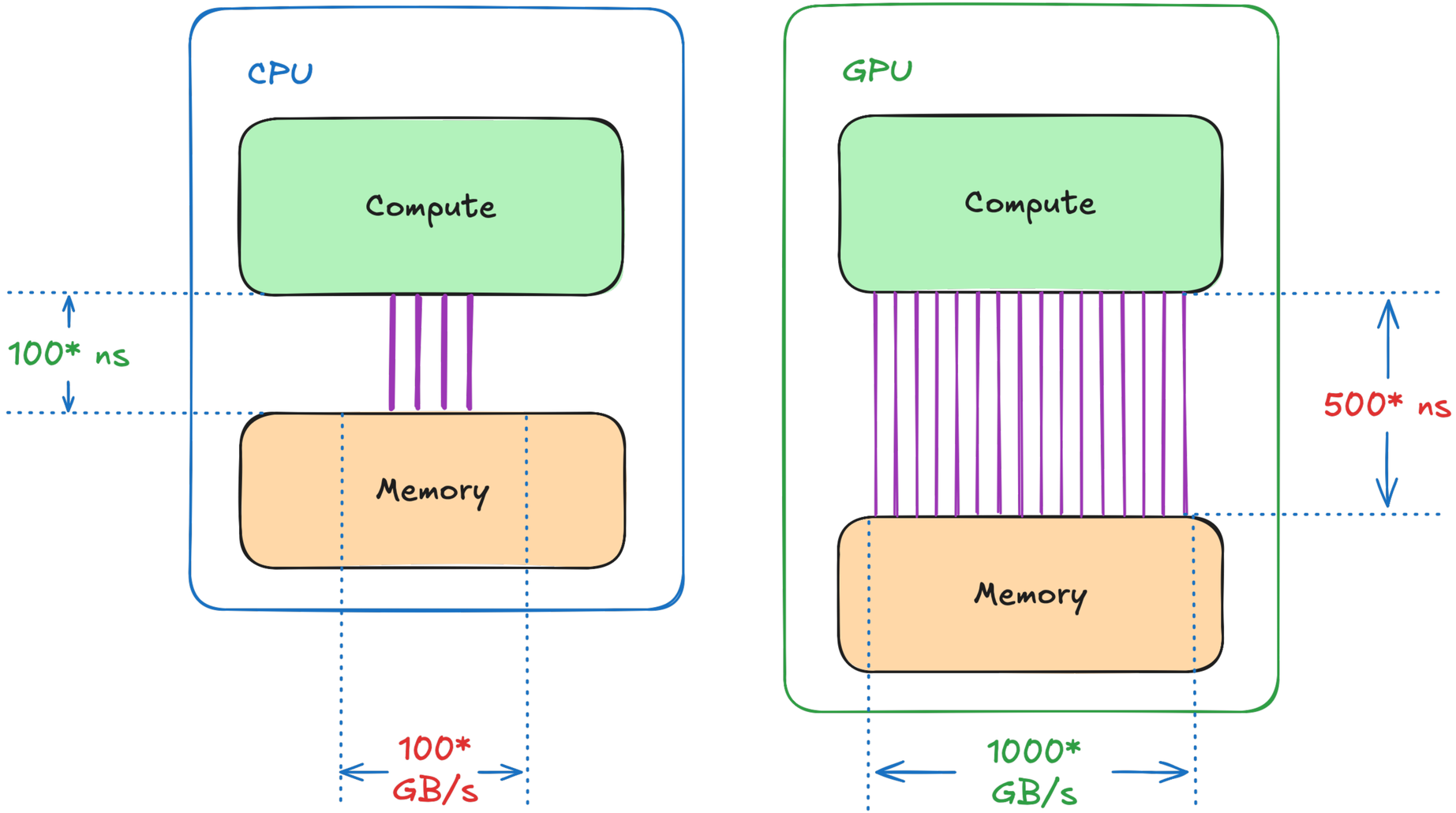
- What's faster? Bus or car?
- The question is not precise enough
- Faster in doing what?

- Bus will take longer to move a few people
- Bus is optimized for moving many people



- Car is optimized for moving a few people
- Car will take longer to move many people

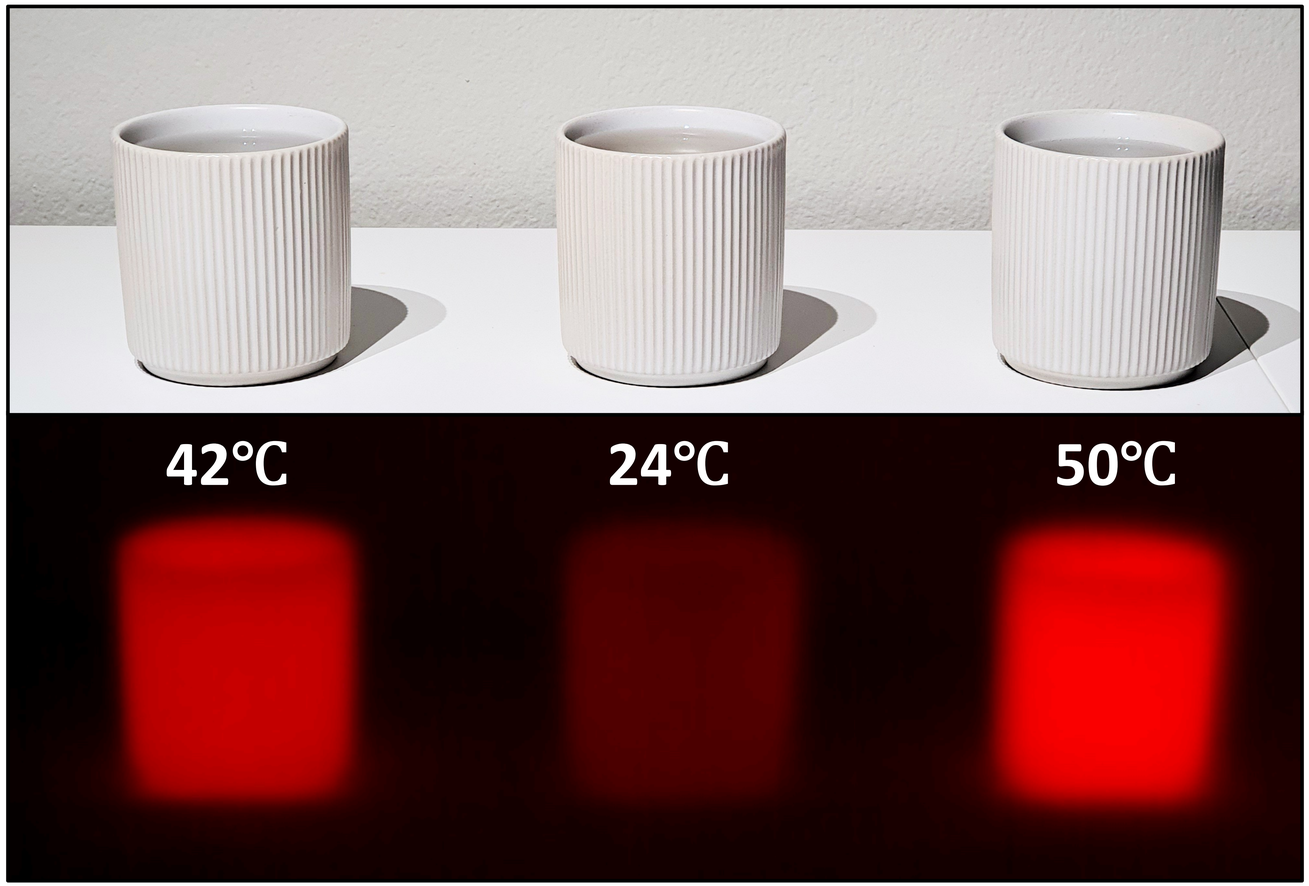
GPU Architecture at a Glance



*numbers are made up

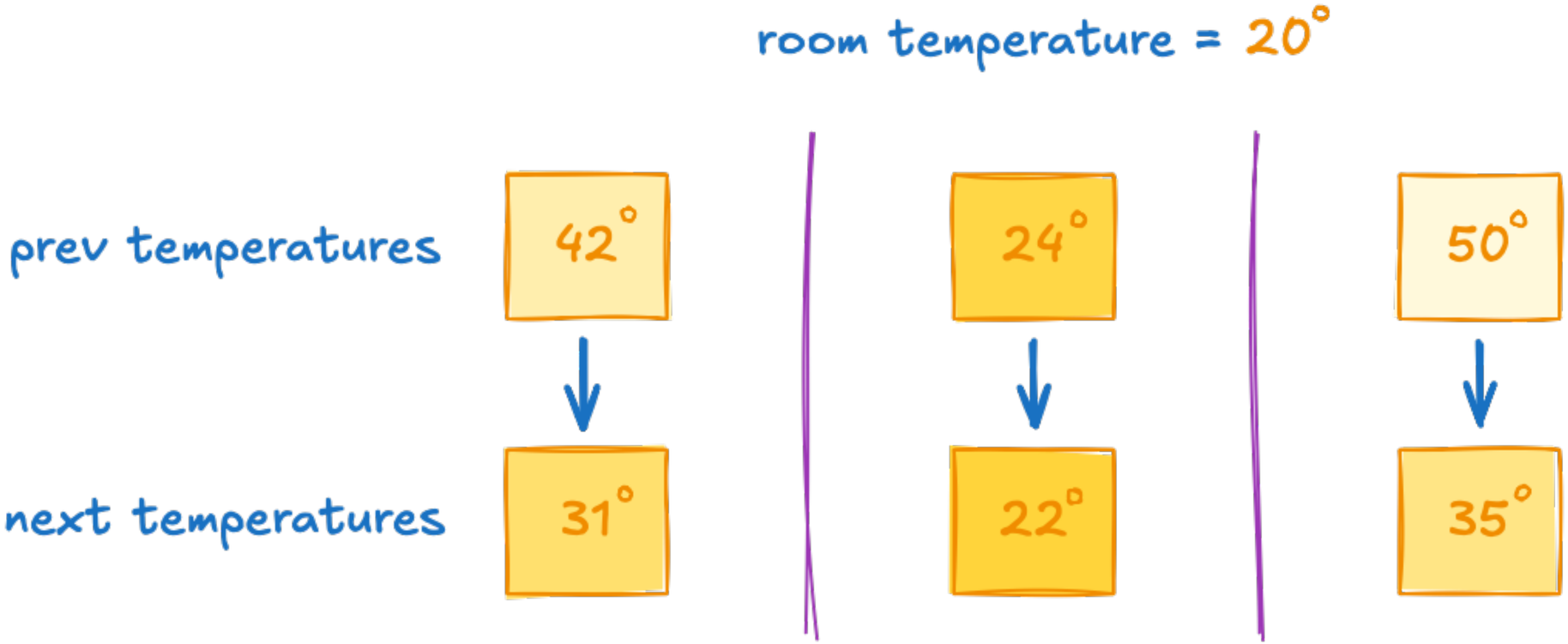
Which Problems Benefit from Bandwidth?

- Let’s say we want to simulate **3** cups cooling down to a room temperature
- We can abstract this problem a bit, by emulating a “sensor” in each cup

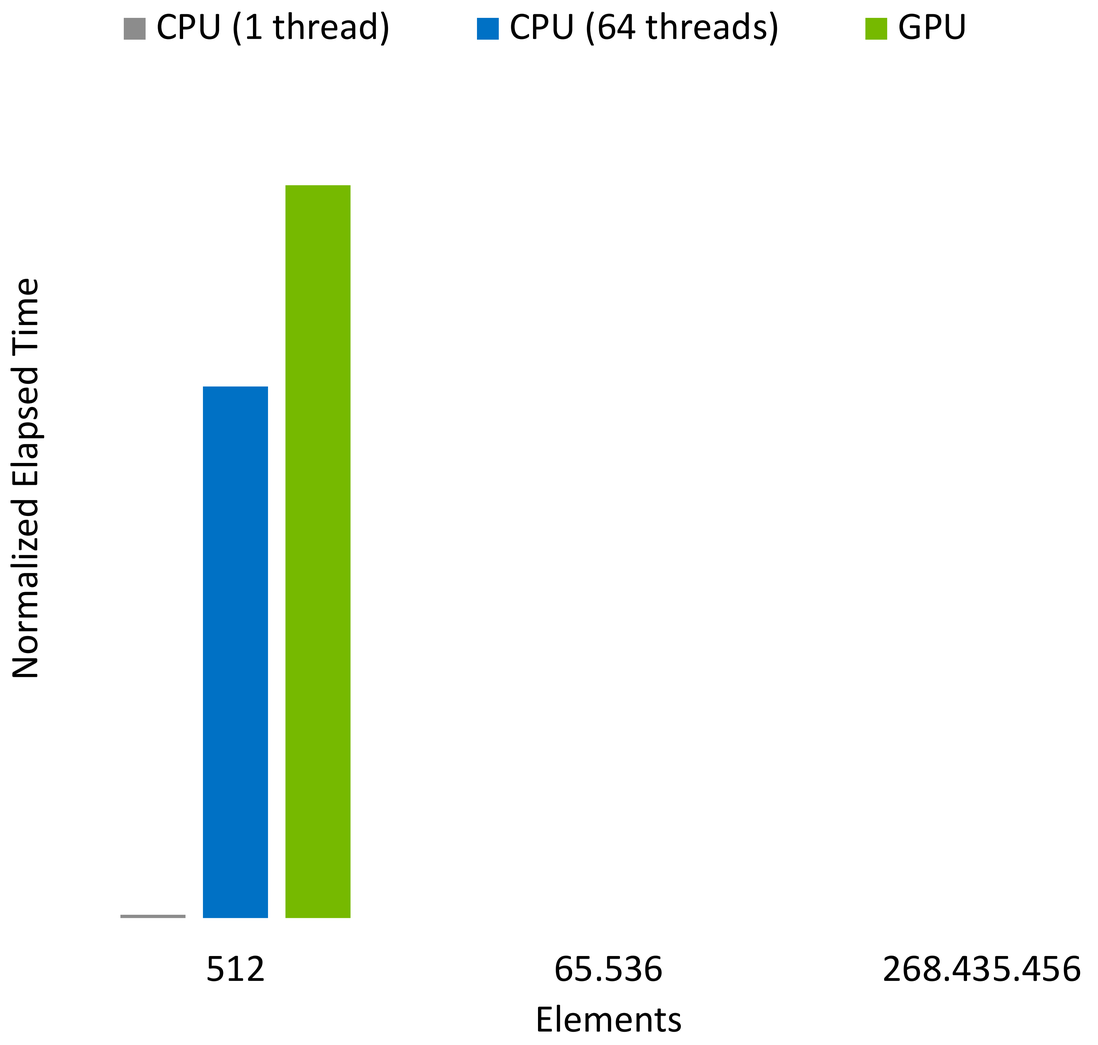


Which Problems Benefit from Bandwidth?

- Let’s say we want to simulate **N** objects cooling down to a room temperature
- Can this problem benefit from immense bandwidth provided by GPUs?
- Just like before, the question is not precise enough
- How large is **N**?

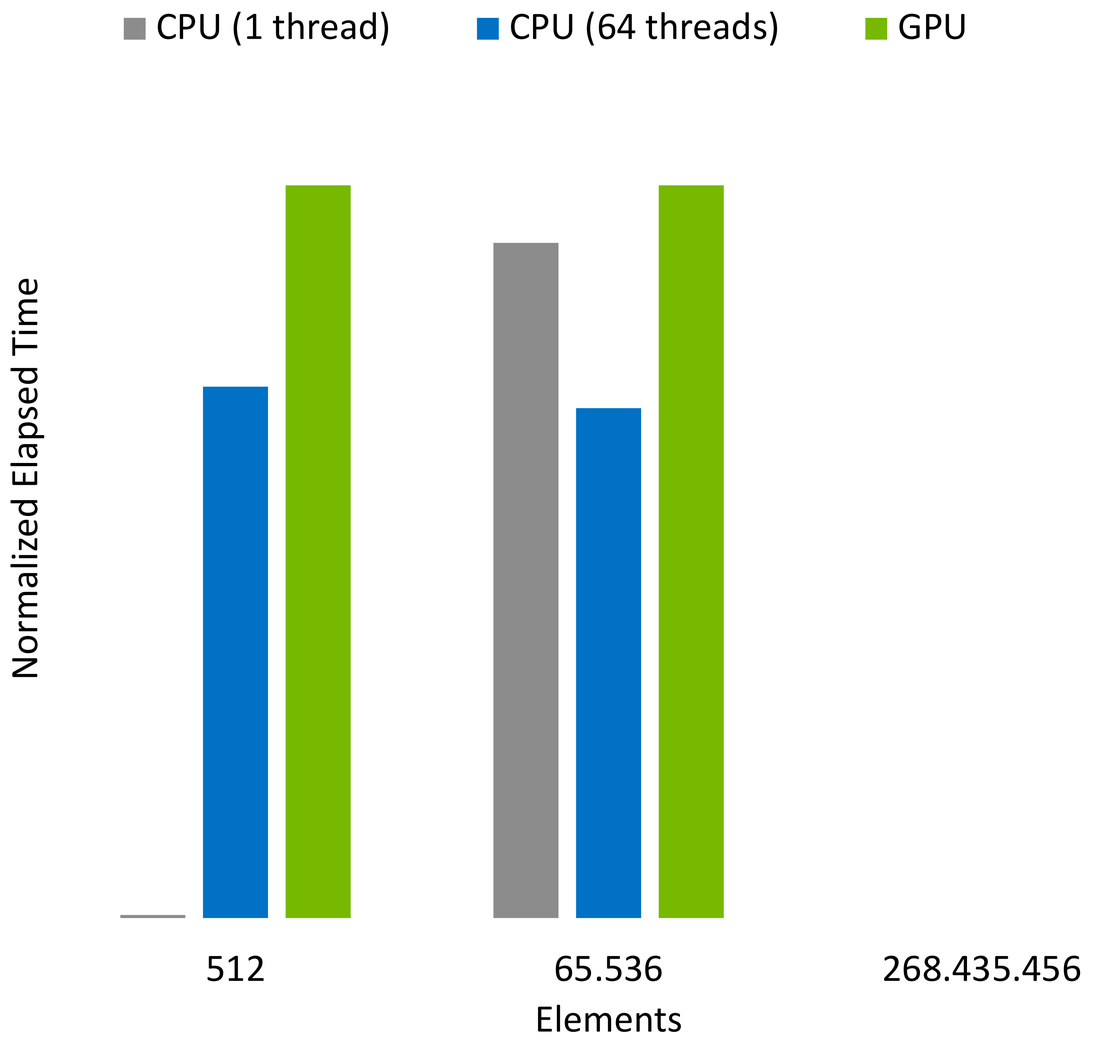


Why GPU Programming?



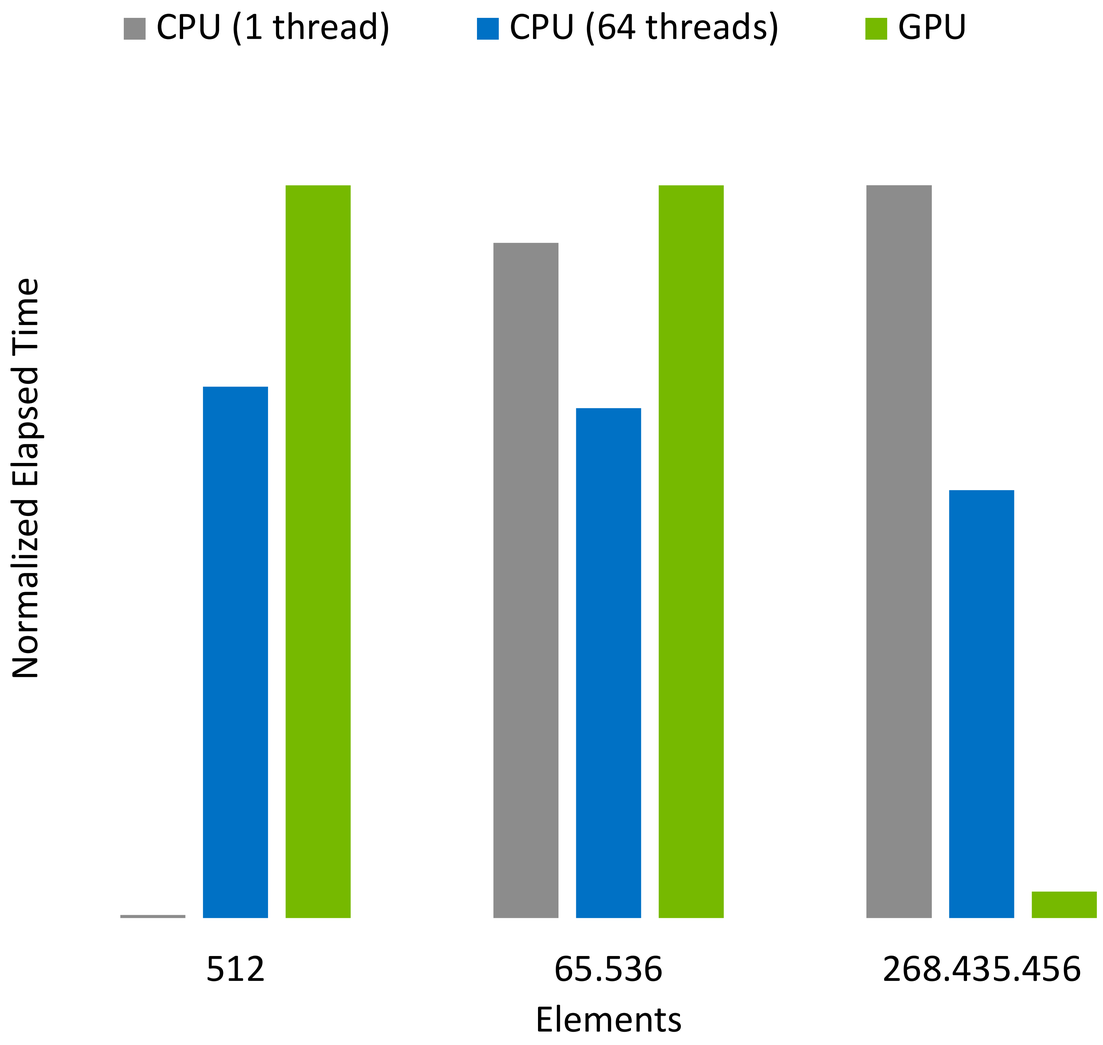
- Single-threaded **CPU** code is 150x faster than multi-threaded one (and 200x faster than **GPU**) when simulating 512 objects

Why GPU Programming?



- Single-threaded **CPU** code is 150x faster than multi-threaded one (and 200x faster than **GPU**) when simulating 512 objects
- But as problem size grows, multi-threaded code overcomes limitations of a single thread, making it a better option when simulating 64k elements

Why GPU Programming?



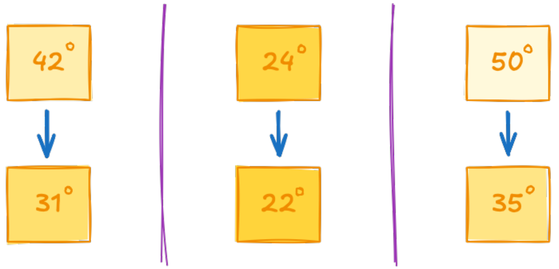
- Single-threaded **CPU** code is 150x faster than multi-threaded one (and 200x faster than **GPU**) when simulating 512 objects
- But as problem size grows, multi-threaded code overcomes limitations of a single thread, making it a better option when simulating 64k elements
- Finally, **GPU** achieves 10x of multi-threaded CPU bandwidth, hence it’s the fasted on 268M elements

Agenda

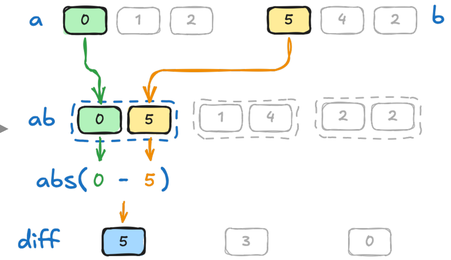
- CUDA Made Easy: Accelerating Applications with Parallel Algorithms
- Lunch Break (30 – 40 mins)
- Unlocking the GPU’s Full Potential: Asynchrony and CUDA Streams
- Break (15 mins)
- Implementing New Algorithms with CUDA Kernels
- Final Assessment (45 minutes)

CUDA Made Easy: Accelerating Applications with Parallel Algorithms

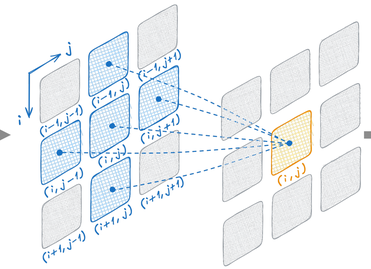
Section at a Glance



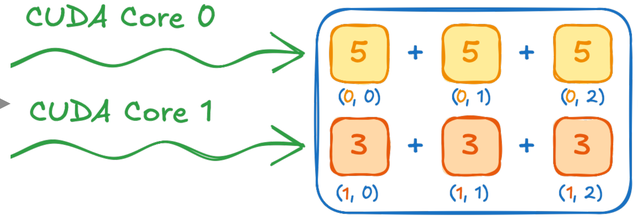
Learn 1.2 **Execution Spaces** as we accelerate naïve cooling simulator and achieve 16x speedup



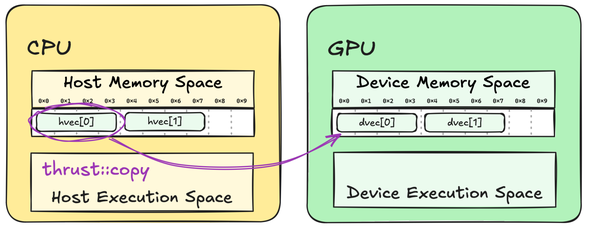
Learn 1.3 **Extending Algorithms** with fancy iterators to achieve 2x speedup



Learn 1.4 **Vocabulary Types** as we transition from naive cooling to stencil-based simulation



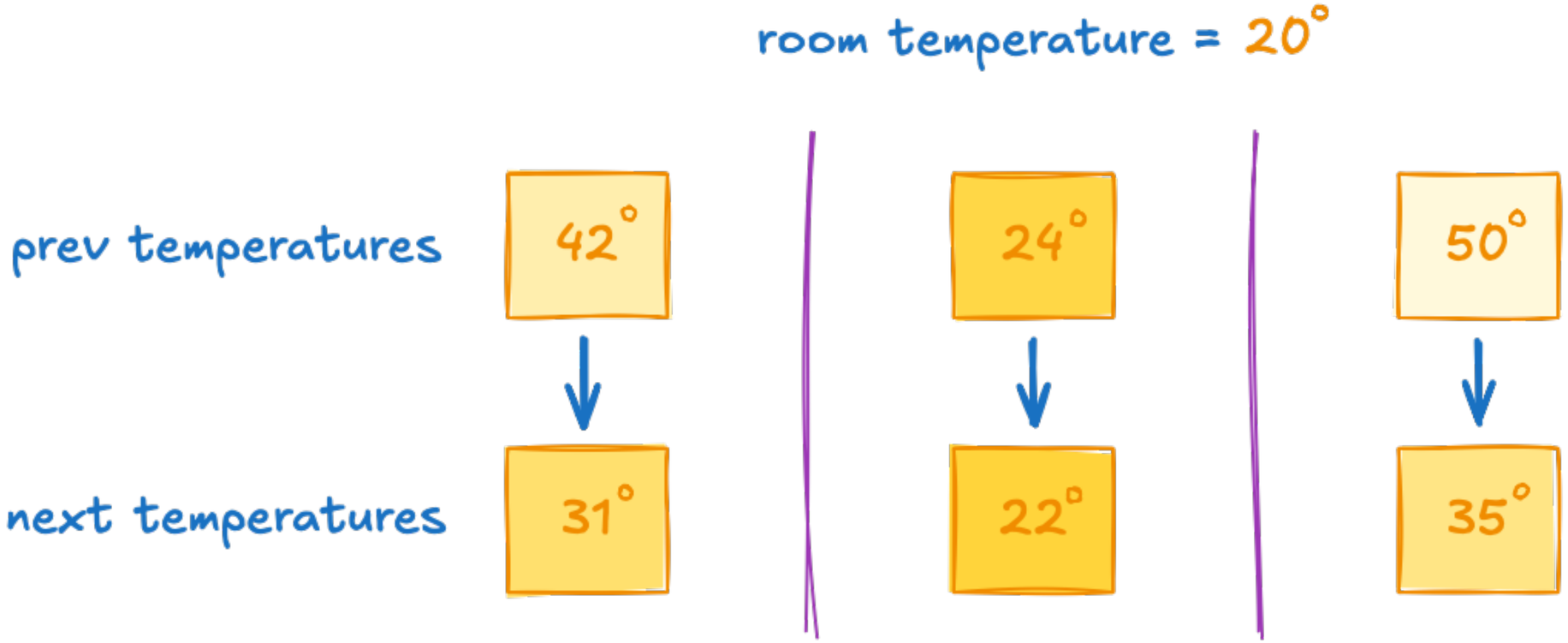
Learn concept of 1.5 **Parallelism** as we accelerate one of the algorithms by 300x



Learn concept of 1.6 **Memory Spaces** as we accelerate one of the algorithms by 100x

Independent Cooling

Problem statement



$$T_n = T_{n-1} + k(T_{room} - T_{n-1})$$

Transform

Simple C++ simulation

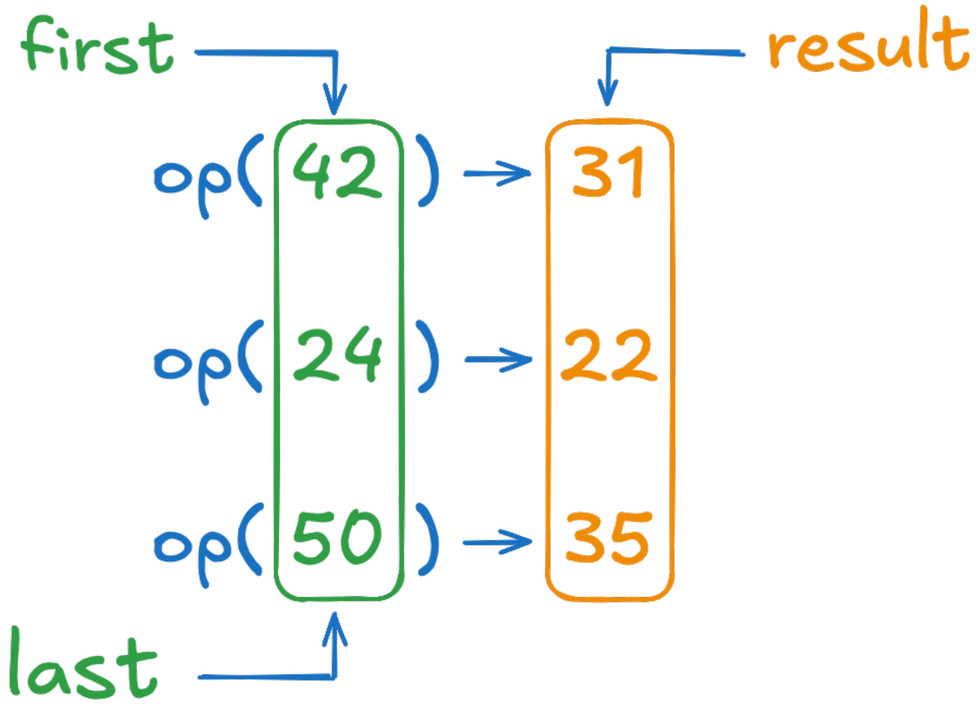
```
float k = 0.5;
float ambient_temp = 20;
std::vector<float> temp{42, 24, 50};

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

for (int step = 0; step < 3; step++)
{
  print(step, temp);
  std::transform(temp.begin(), temp.end(),
                 temp.begin(), op);
}
```

- Multiple objects cool down independently
- Iterative change of state
- Computed on CPU (for now)

step	temp[0]	temp[1]	temp[2]
0	42.0	24.0	50.0
1	31.0	22.0	35.0
2	25.5	21.0	27.5

Transform

For execution on GPU



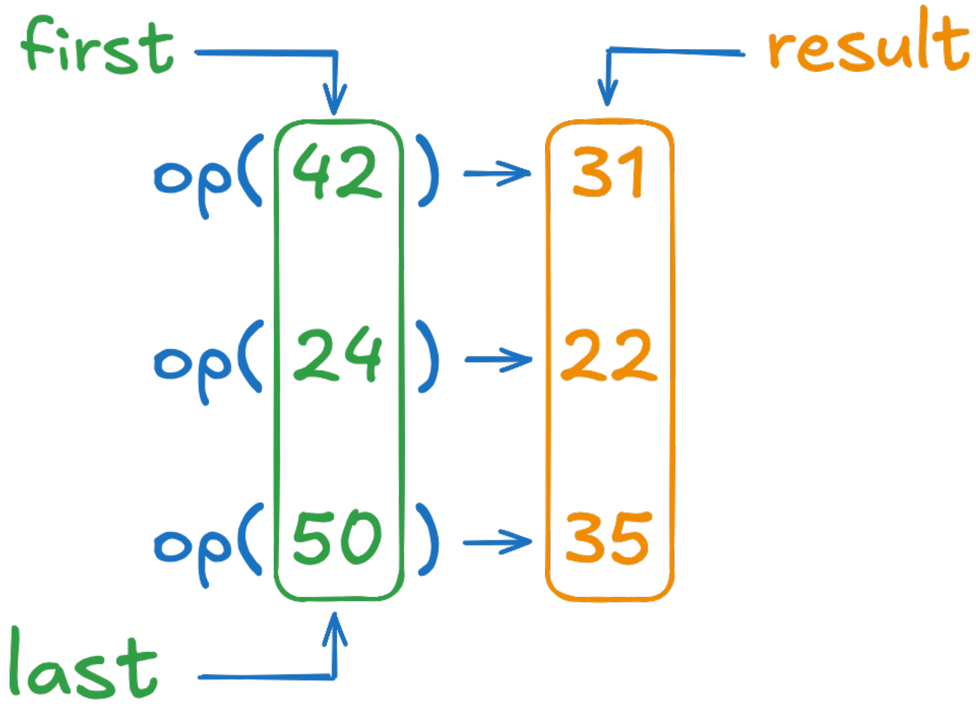

```
auto op = [=](float temp) {
  float diff = ambient_temp - temp;
  return temp + k * diff;
};

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

- applies a given function to the elements of the input range, and
- stores the result in the output range

std::transform(first , last , result , op)

Transform

For execution on GPU

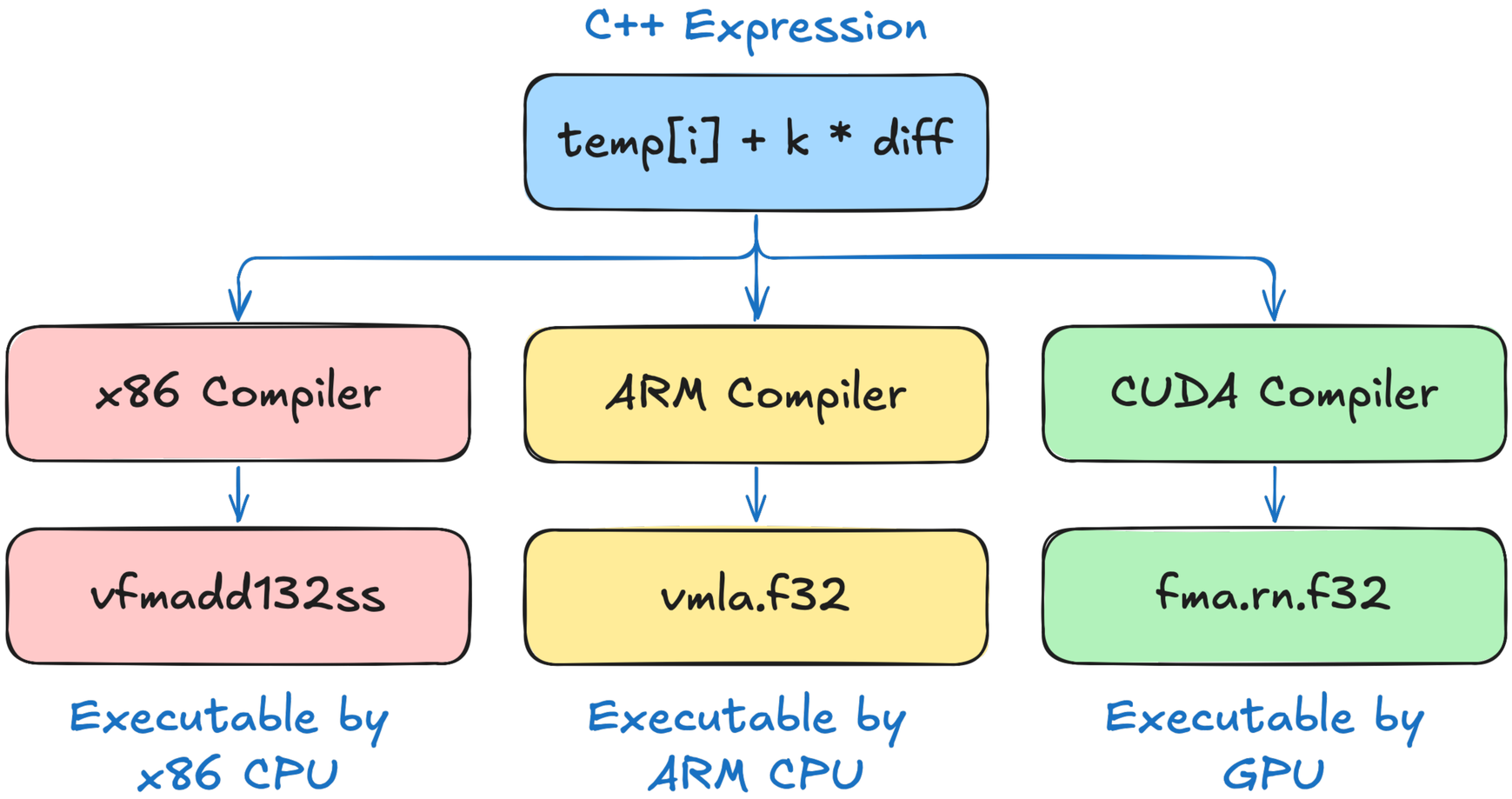



```
auto op = [=](float temp) {
  float diff = ambient_temp - temp;
  return temp + k * diff;
};

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

for (int i = 0; i < temp.size(); i++) {
  temp[i] = op(temp[i]);
}
```

std::transform(first , last , result , op)

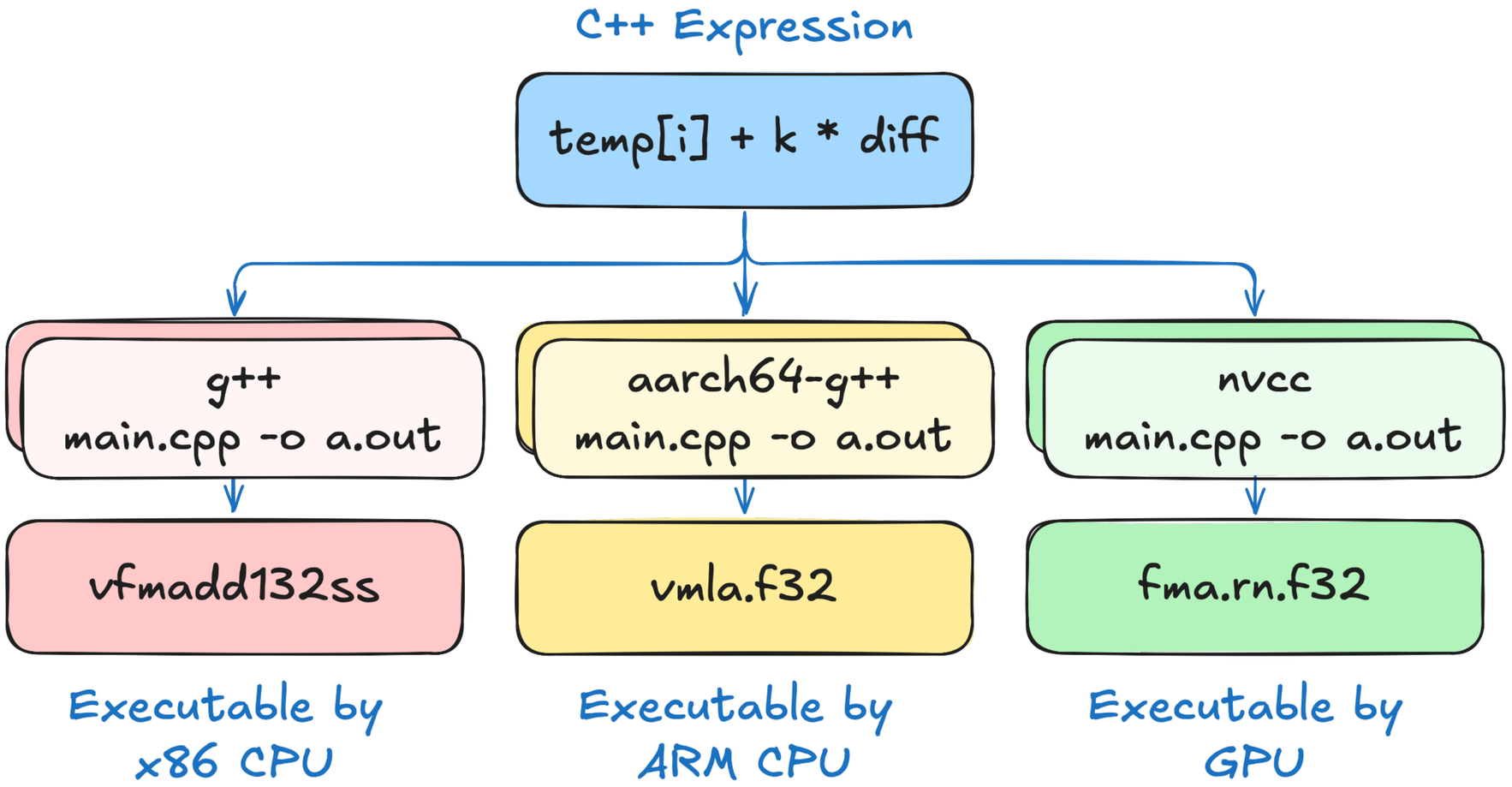
Compilation

Of a simple C++ simulation



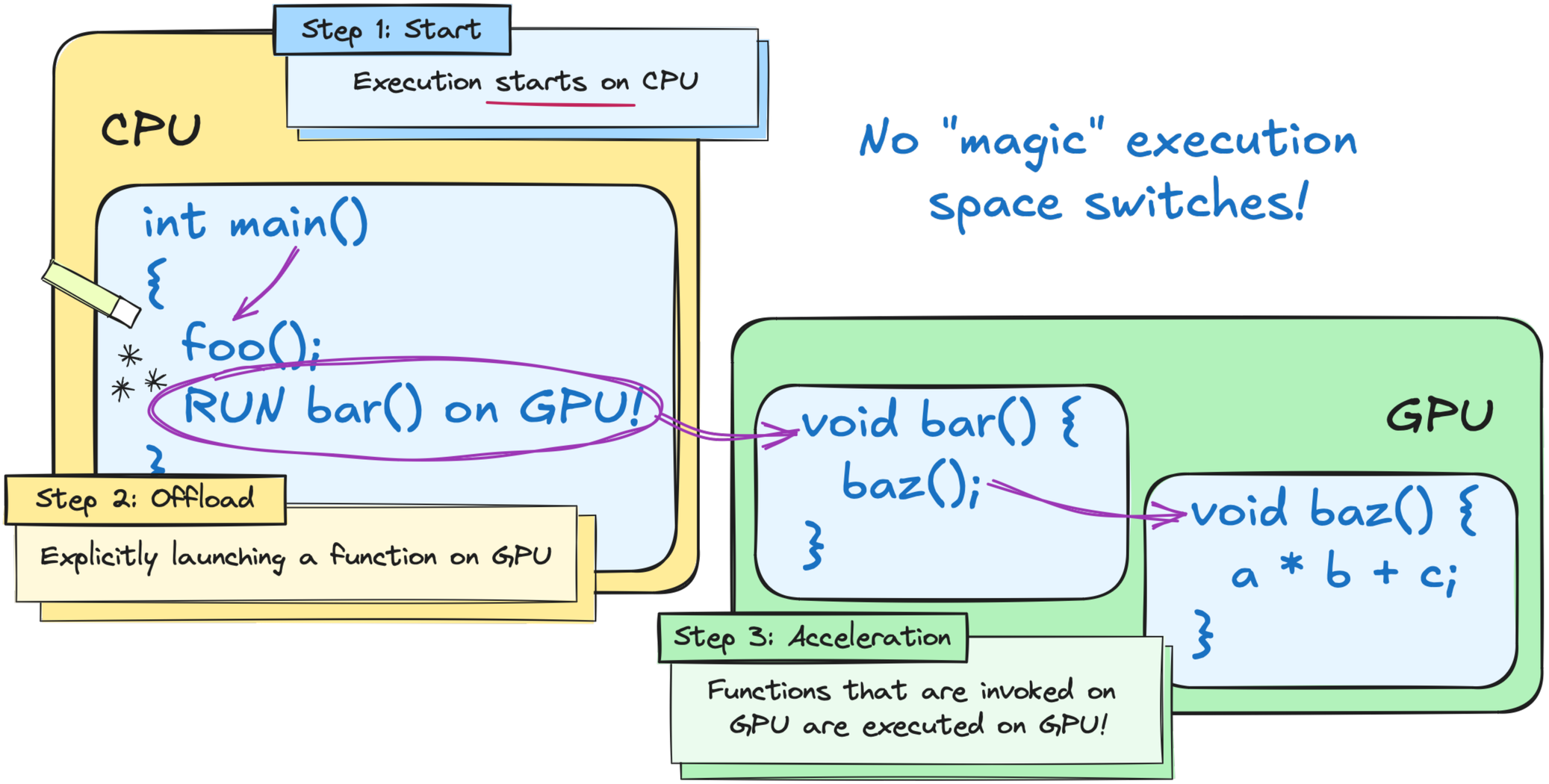
Compilation

Of a simple C++ simulation

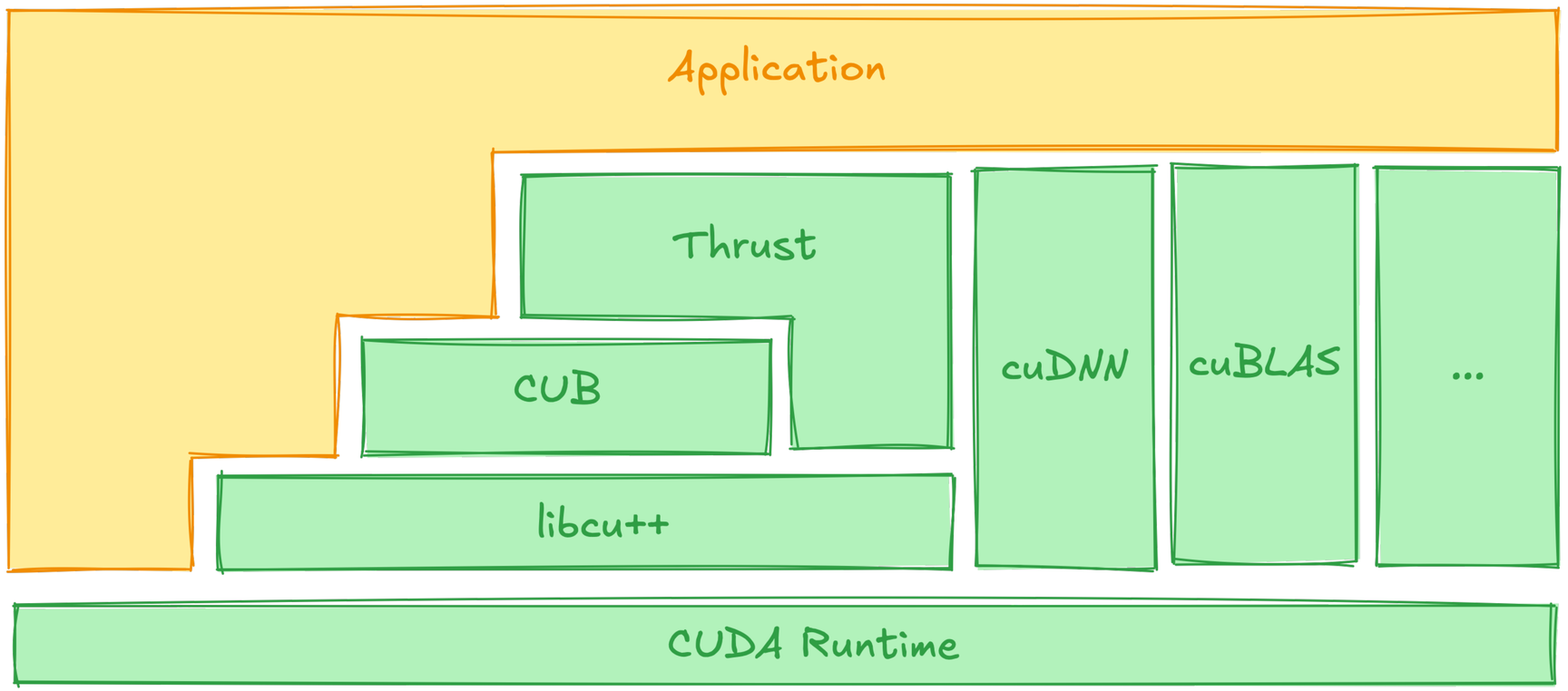


- NVCC (NVIDIA CUDA Compiler Driver) knows how to turn C++ code into GPU instructions

Heterogeneous Programming Model



Software Stack



- Applications utilize libraries to:
  - simplify development
  - maximize performance
- CUDA Runtime
  - Underpins the entire stack
  - Provides interface to GPU

Thrust Overview

Standard Algorithms

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

Containers

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

Function Objects

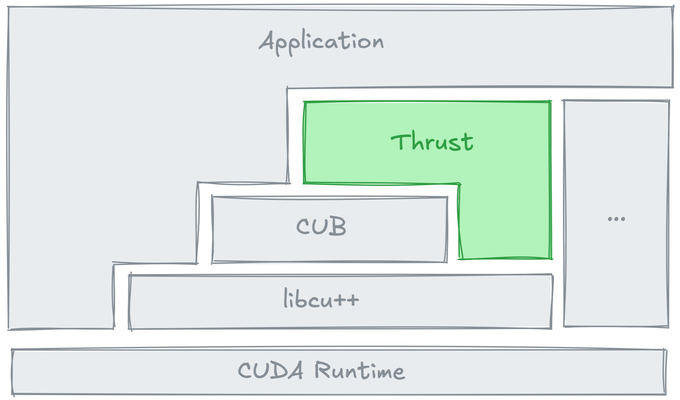
- thrust::plus
- thrust::less
- thrust::multiplies
- ...

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

Porting Algorithms to GPU

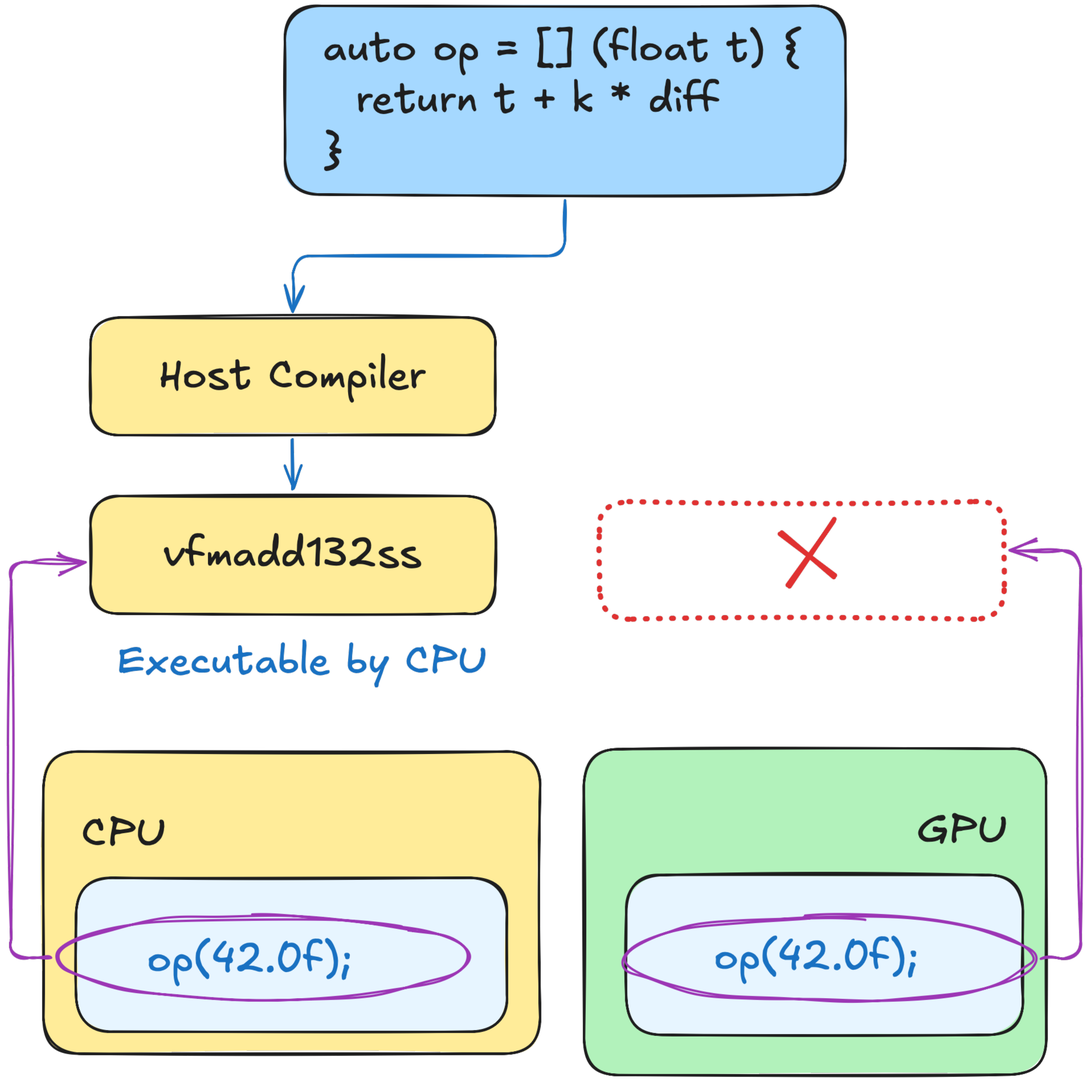
Using parallel algorithms

```
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);
```

Porting Algorithms to GPU

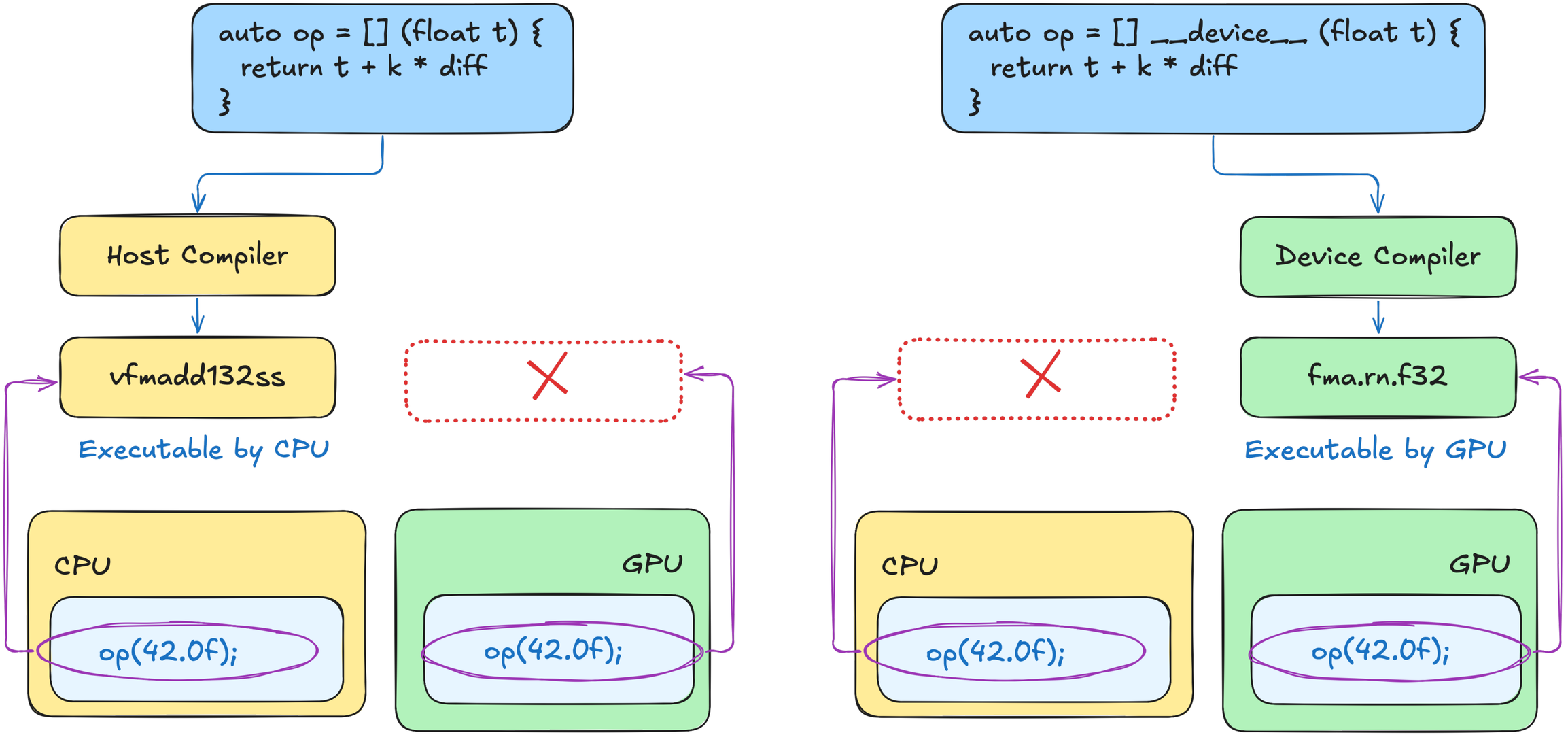
Execution space specifier



- By default, all functions are compiled for CPU
- If you attempt to invoke this function on GPU, there'll be no code for it to execute

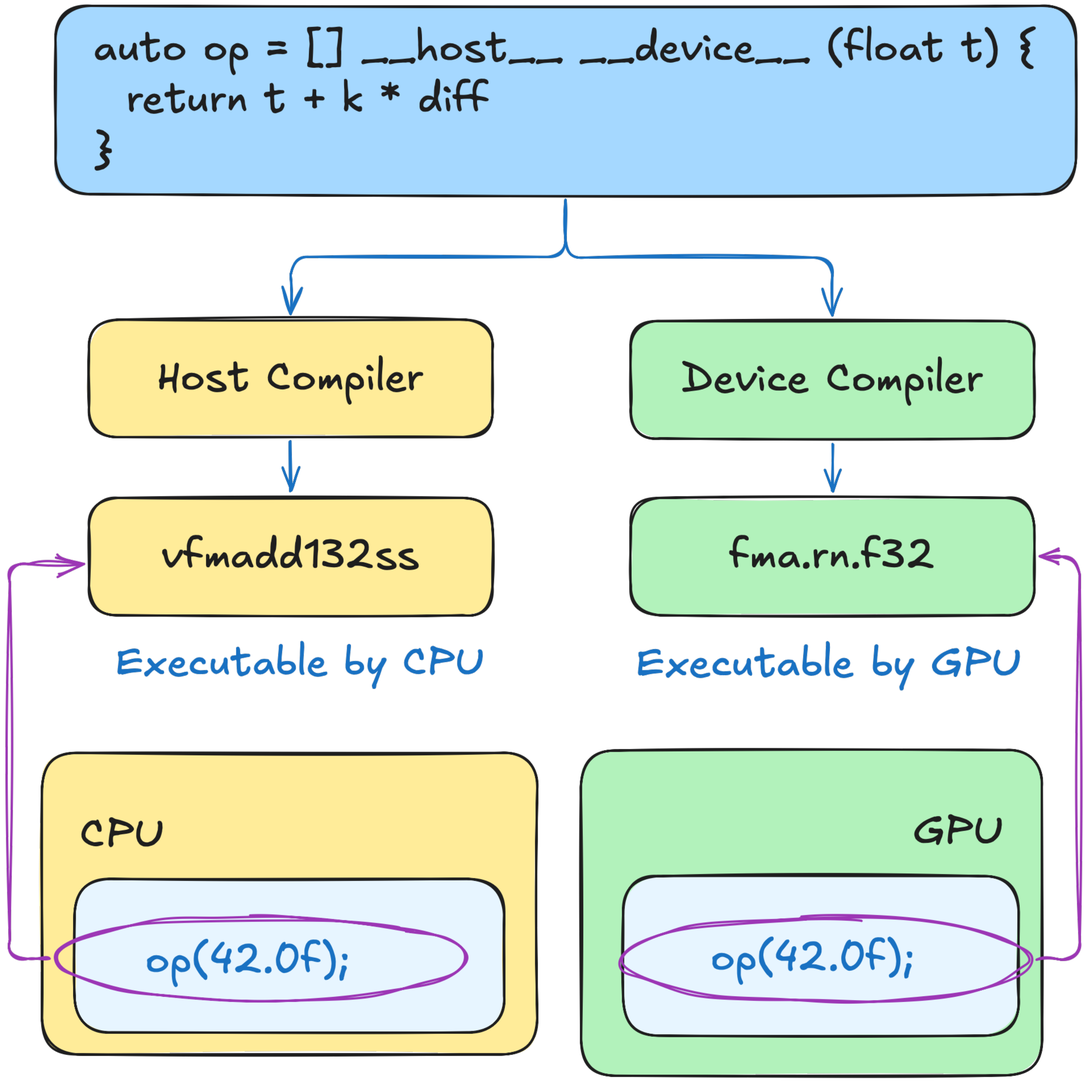
Porting Algorithms to GPU

Execution space: host & device



Porting Algorithms to GPU

Execution space: host-device




```
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);
```

Execution Policy

C++

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

CUDA C++

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

What is an execution policy for?

- It tells Thrust *where* to run the algorithm

thrust::host

- Executes algorithms on the **CPU** (host)

Consistent interface

- Allows code reuse between CPU and GPU

thrust::device

- Executes algorithms on the **GPU** (device).

Execution Policy is Not New

C++

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

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

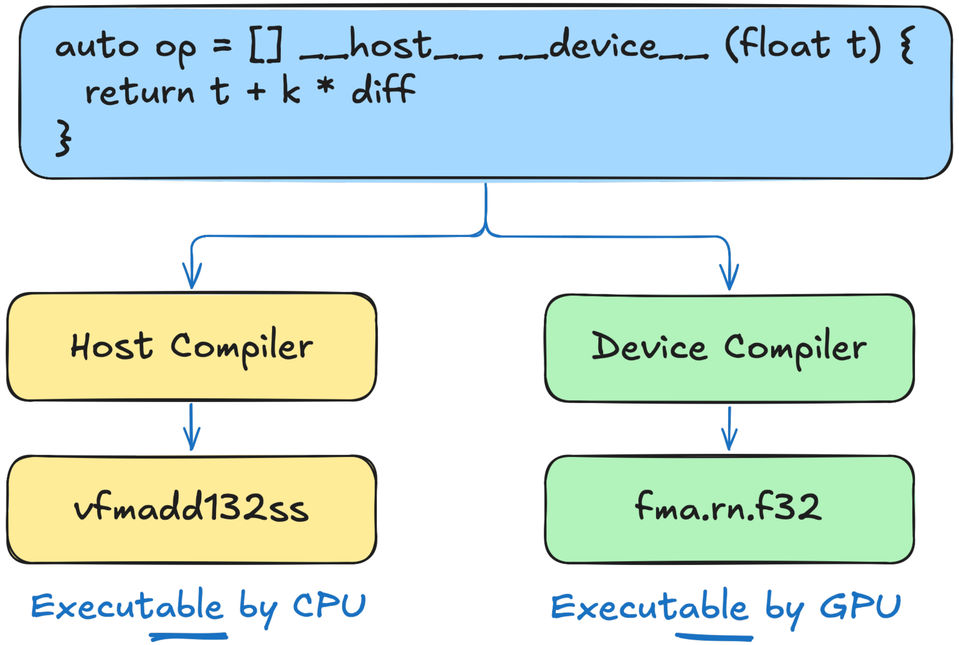
CUDA C++

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

- Execution policy is not a Thrust-specific concept
- If you are familiar with C++ 17 parallel algorithms, Thrust execution policies are like std::execution

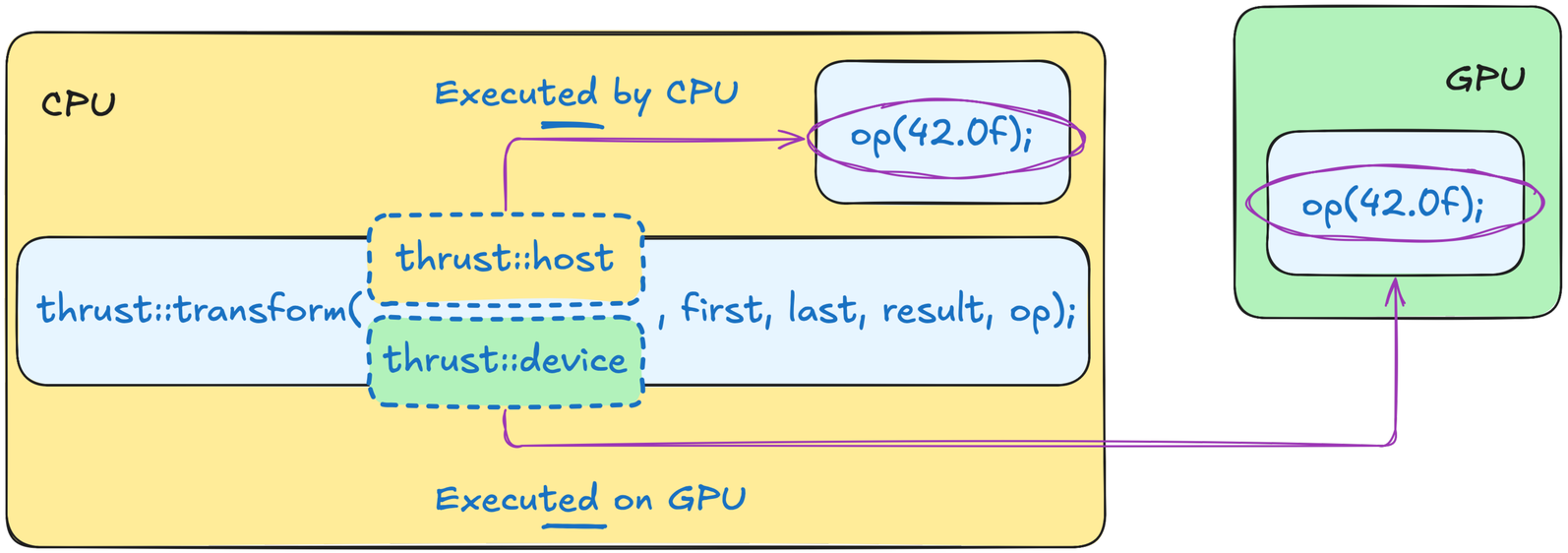
Execution Policy vs Specifier

- execution space **specifier** (__host__, __device__):
  - works at compile time
  - indicates where code **can run**
  - doesn't automatically run code there



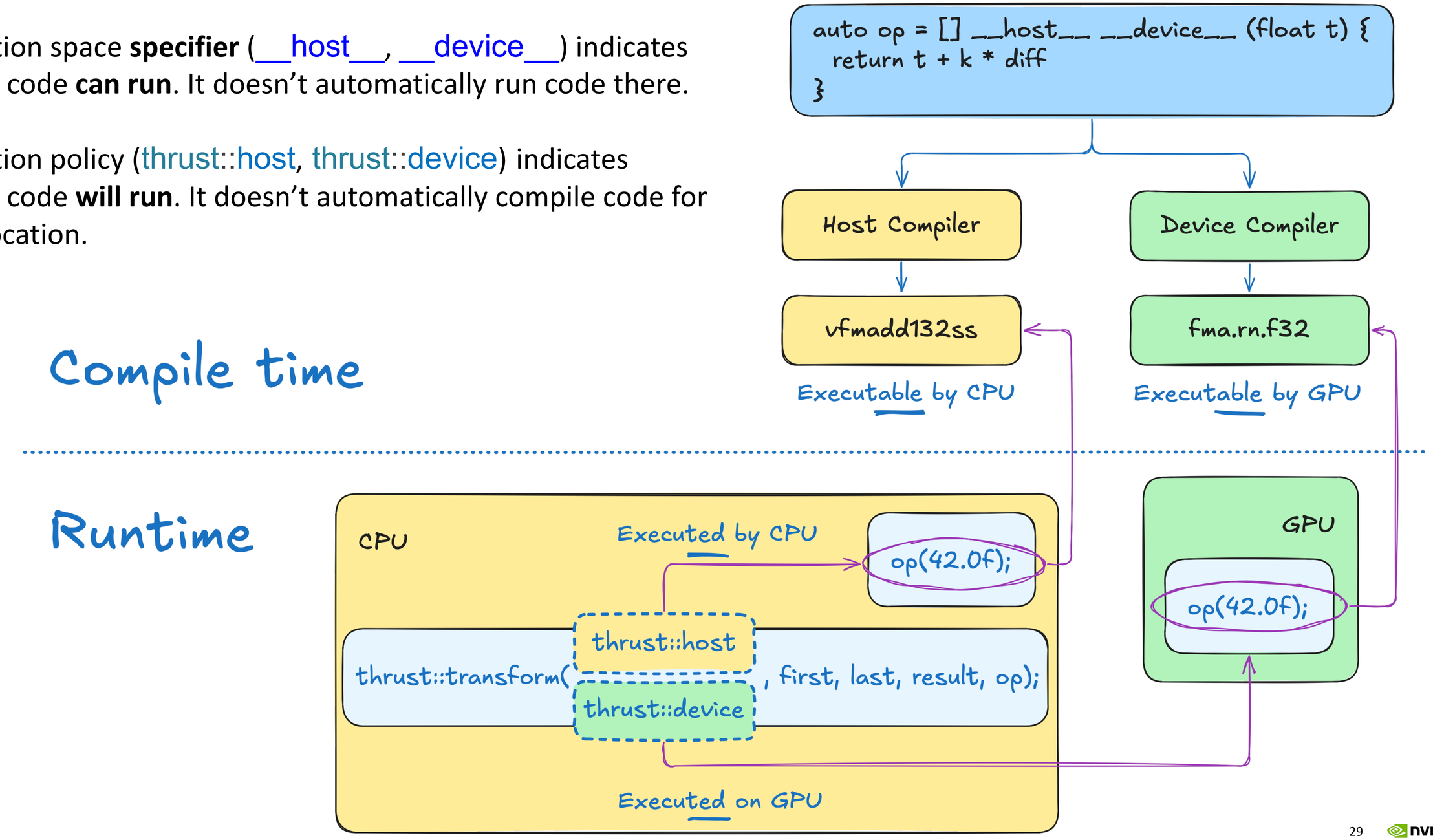
Execution Policy vs Specifier

- execution policy (thrust::host, thrust::device):
  - works at runtime
  - indicates where code **will run**
  - doesn't automatically compile code for that location.



Execution Policy vs Specifier

- execution space **specifier** (__host__, __device__) indicates where code **can run**. It doesn't automatically run code there.
- execution policy (thrust::host, thrust::device) indicates where 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**

Exercise: Annotate Execution Spaces

5 minutes

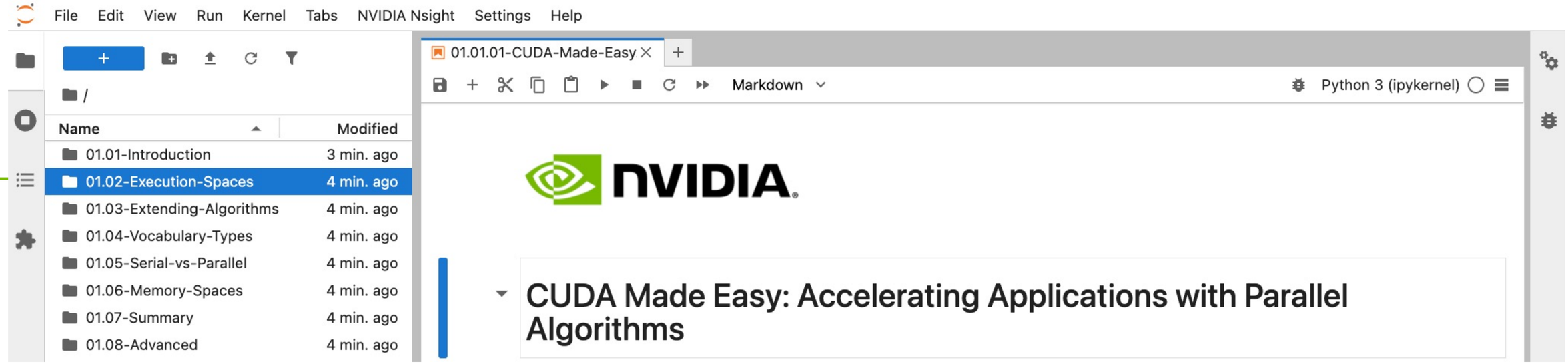
- Replace ??? with CPU or GPU

```
dli::where_am_I("???");

thrust::universal_vector<int> vec{1};
thrust::for_each(thrust::device, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("???"); });

thrust::for_each(thrust::host, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("???"); });

dli::where_am_I("???");
```

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

How to Run Exercises

Locate the exercise folder



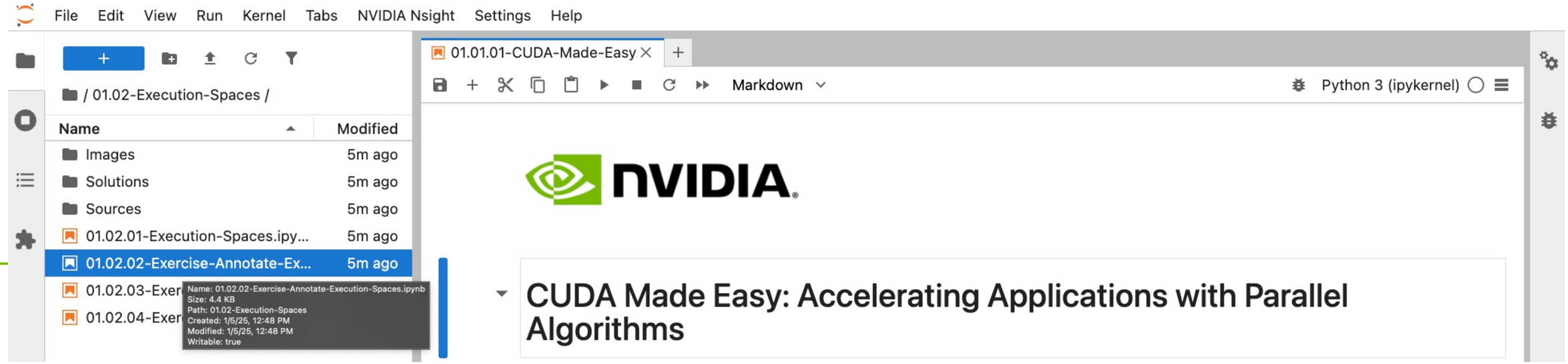
[] __host__ __device__(int) { dli::where_am_I("???"); });

dli::where_am_I("???");

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

How to Run Exercises

Locate the notebook



[] __host__ __device__(int) { dli::where_am_I("???"); });

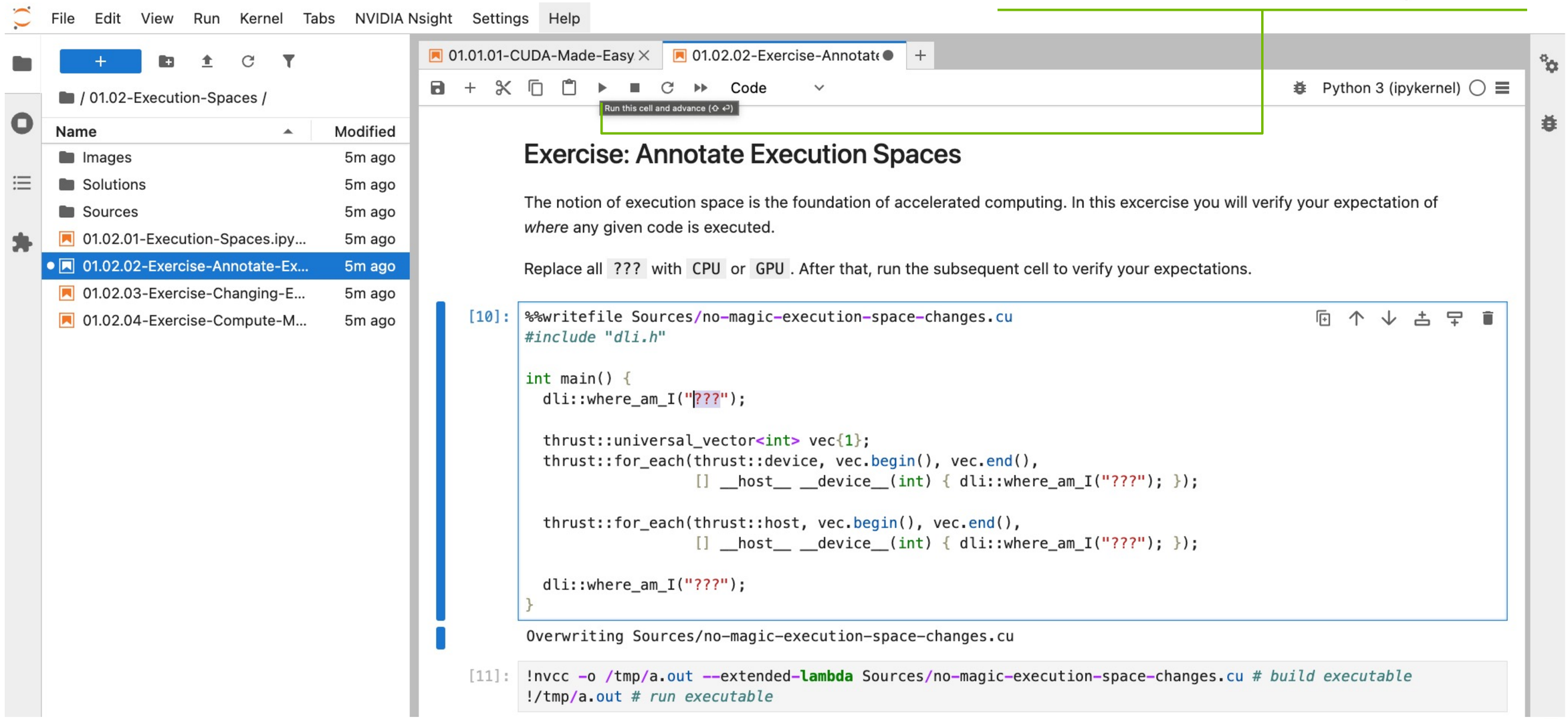
dli::where_am_I("???");

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

How to Run Exercises

Saving code changes

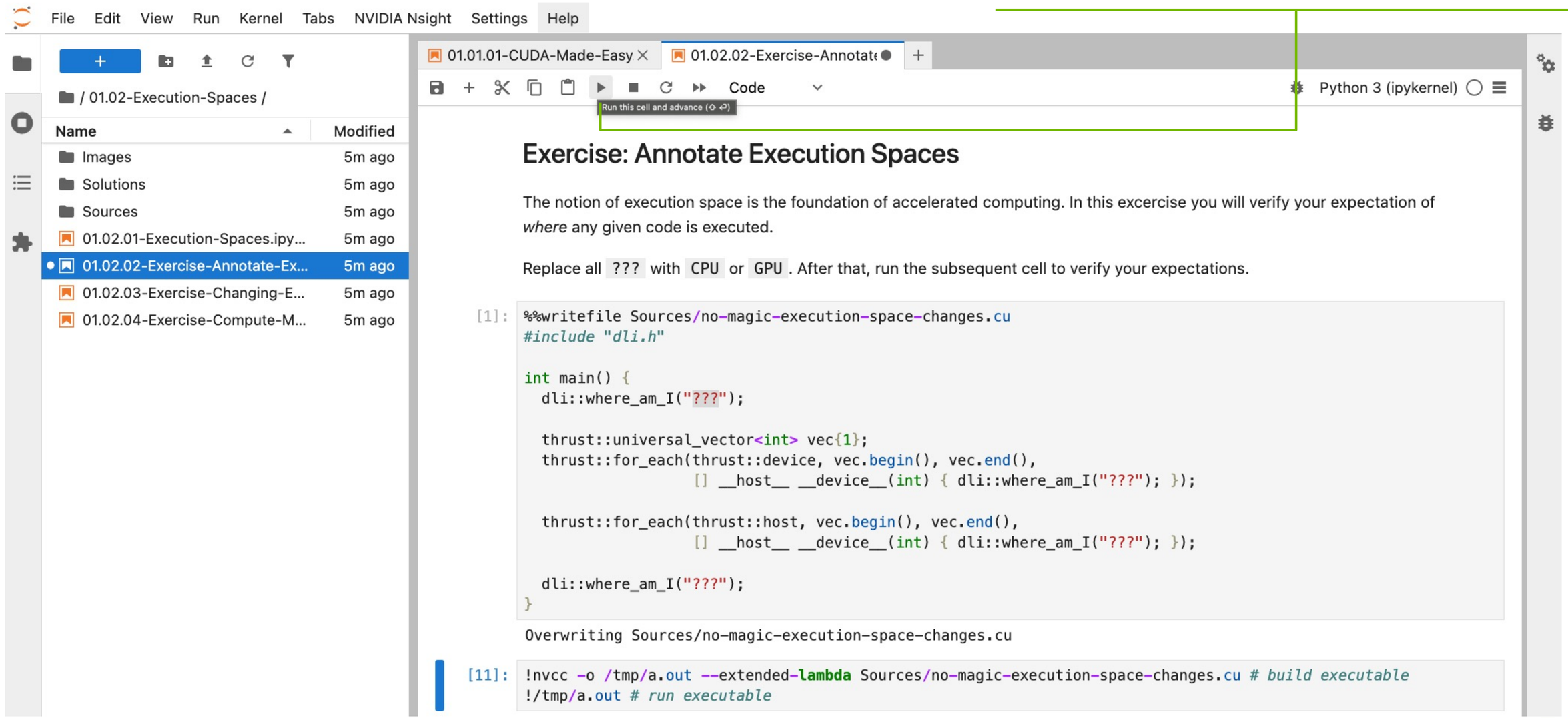
Run the cell to save changes



How to Run Exercises

Run the code

Run the cell to execute the code



Exercise: Annotate Execution Spaces

Solution

dli::where_am_I("**CPU**");

Program always starts on **CPU**

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

Exercise: Annotate Execution Spaces

Solution

```
dli::where_am_I("CPU");

thrust::universal_vector<int> vec{1};
thrust::for_each(thrust::device, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("GPU"); });
```

thrust::device execution policy leads to execution on **GPU**

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

Exercise: Annotate Execution Spaces

Solution

```
dli::where_am_I("CPU");

thrust::universal_vector<int> vec{1};
thrust::for_each(thrust::device, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("GPU"); });

thrust::for_each(thrust::host, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("CPU"); });
```

thrust::host execution policy leads to execution on **CPU**

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

Exercise: Annotate Execution Spaces

Solution

```
dli::where_am_I("CPU");

thrust::universal_vector<int> vec{1};
thrust::for_each(thrust::device, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("GPU"); });

thrust::for_each(thrust::host, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { dli::where_am_I("CPU"); });

dli::where_am_I("CPU");
```

Device execution space doesn't "leak" out of algorithms

01.02-Execution-Spaces/01.02.02-Exercise-Annotate-Execution-Spaces.ipynb

Exercise: Change Execution Space

5 minutes

- Modify Thrust algorithm to run on GPU

```
thrust::for_each(thrust::host, vec.begin(), vec.end(), [] __host__ (int val) {
  std::printf("printing %d on %s\n", val, dli::execution_space());
});
```

01.02-Execution-Spaces/01.02.03-Exercise-Changing-Execution-Space.ipynb

Exercise: Change Execution Space

Solution

```
thrust::for_each(thrust::host, vec.begin(), vec.end(), [] __host__ (int val) {
  std::printf("printing %d on %s\n", val, dli::execution_space());
});
```

Use thrust::device to tell Thrust that you want algorithm *to run* on GPU

```
thrust::for_each(thrust::device, vec.begin(), vec.end(), [] __device__(int val) {
  std::printf("printing %d on %s\n", val, dli::execution_space());
});
```

Exercise: Change Execution Space

Solution

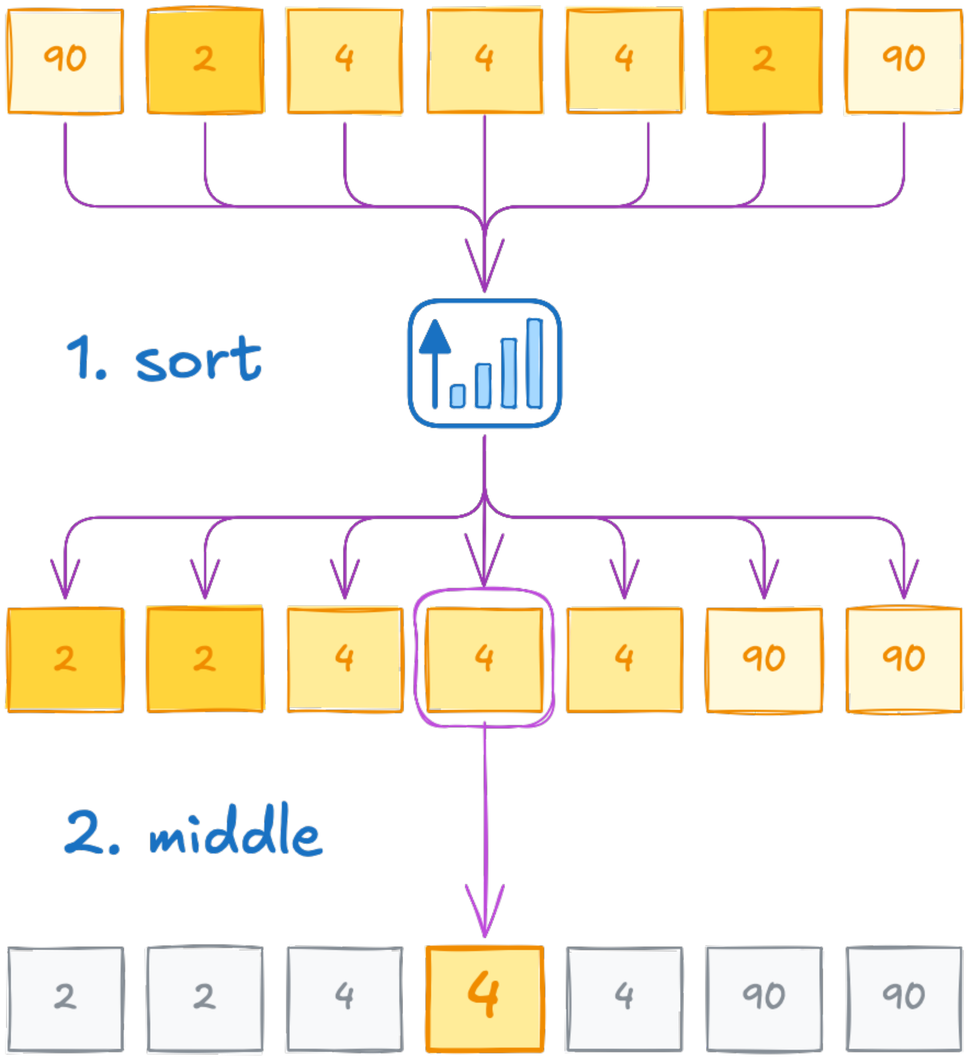
```
thrust::for_each(thrust::host, vec.begin(), vec.end(), [] __host__ (int val) {
  std::printf("printing %d on %s\n", val, dli::execution_space());
});
```

Use __device__ to tell NVCC that you need a function *to be compiled* for GPU

```
thrust::for_each(thrust::device, vec.begin(), vec.end(), [] __device__(int val) {
  std::printf("printing %d on %s\n", val, dli::execution_space());
});
```

Median

CPU version



- Median is a number which is larger and less than half of elements
- To compute median, we can sort the data and then return the middle element
- For simplicity, we'll only cover odd number of elements

```
float median(thrust::universal_vector<float> vec)
{
  std::sort(vec.begin(), vec.end());
  return vec[vec.size() / 2];
}
```

Exercise: Port to GPU

5 minutes

- Modify code to GPU-accelerate median computation

```
float median(thrust::universal_vector<float> vec)
{
  std::sort(vec.begin(), vec.end());
  return vec[vec.size() / 2];
}

int main()
{
  for (int step = 0; step < 3; step++)
  {
    float median_temp = median(temp);
    ...
  }
}
```

01.02-Execution-Spaces/01.02.04-Exercise-Compute-Median-Temperature.ipynb

Exercise: Port to GPU

Wrong solution

```
__device__
float median(thrust::universal_vector<float> vec)
{
  std::sort(vec.begin(), vec.end());
  return vec[vec.size() / 2];
}

int main()
{
  for (int step = 0; step < 3; step++)
  {
    float median_temp = median(temp);
    ...
  }
}
```

- Execution specifier doesn't make the function run on **GPU**, only compile code for it
- No reason to **GPU**-accelerate a single division
- Parallel algorithms are invoked from **CPU**

Exercise: Port to GPU

Solution

```
float median(...)
{
  std::sort(vec.begin(), vec.end());
  return vec[vec.size() / 2];
}
```

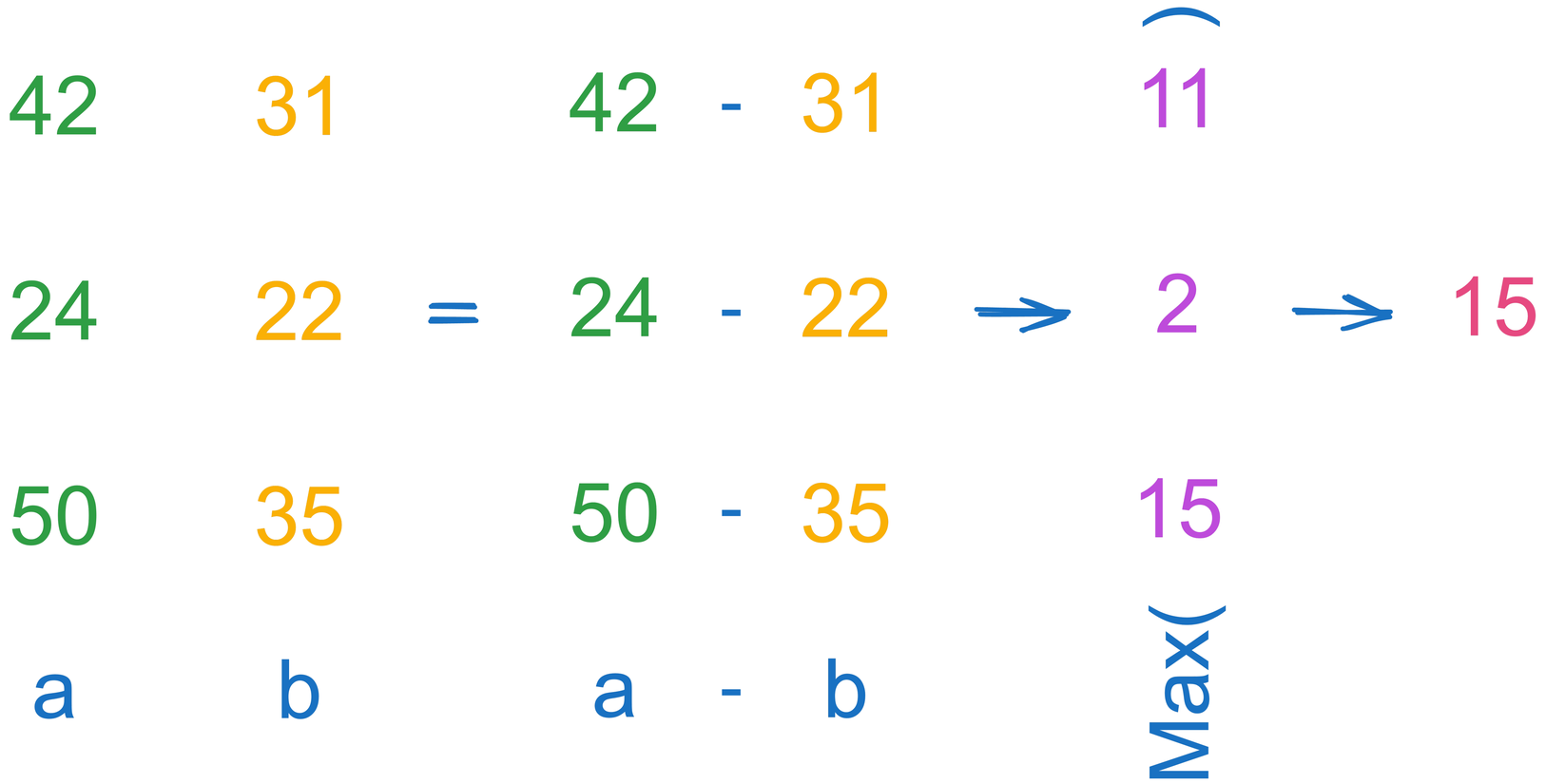
```
float median(...)
{
  thrust::sort(thrust::device, vec.begin(), vec.end());
  return vec[vec.size() / 2];
}
```

GPU-accelerating standard algorithms should be as easy as:

- switching std:: to thrust::, and
- adding an execution policy

Extending Standard Algorithms

Compute max difference between two vectors



There’s no standard algorithm for every use case

Extending Standard Algorithms

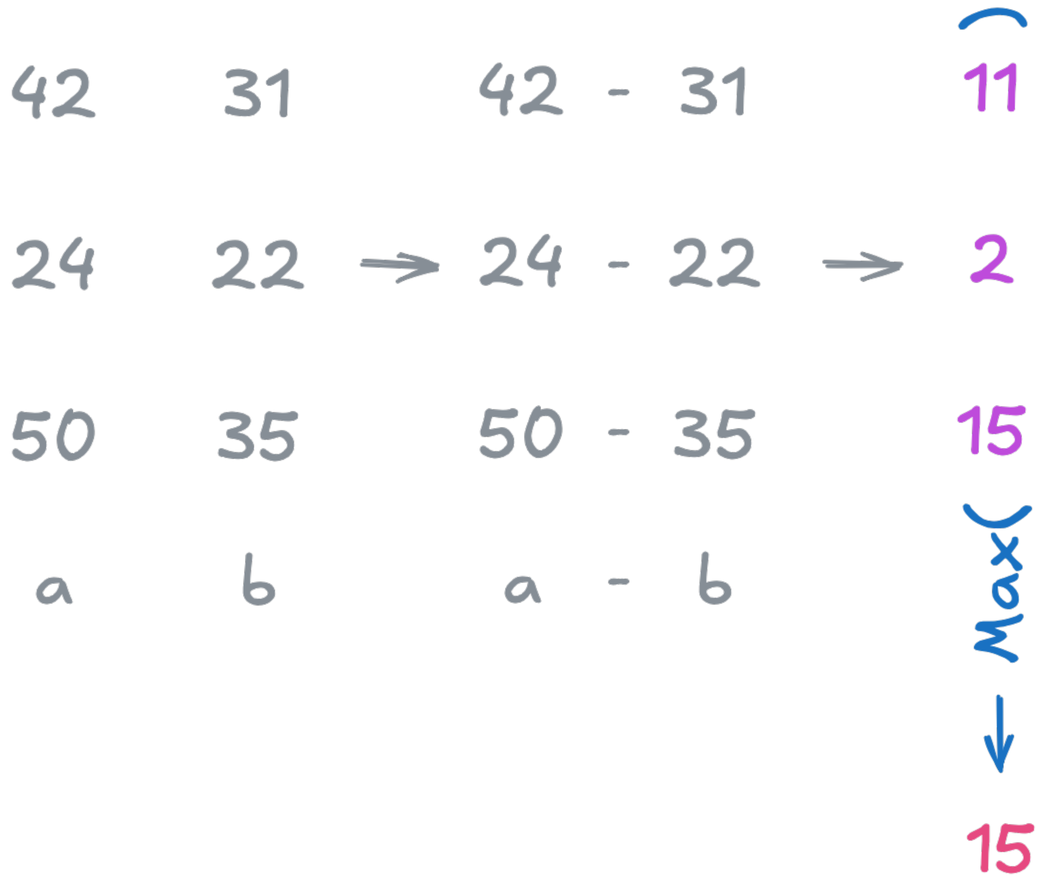
Compute max difference between two vectors

```
thrust::universal_vector<float>
  unnecessarily_materialized_diff(a.size());

// compute abs differences
thrust::transform(
  thrust::device,
  a.begin(), a.end(), // first input sequence
  b.begin(),          // second input sequence
  unnecessarily_materialized_diff.begin(),
  []__host__ __device__(float x, float y) {
    return abs(x - y);
  });
```

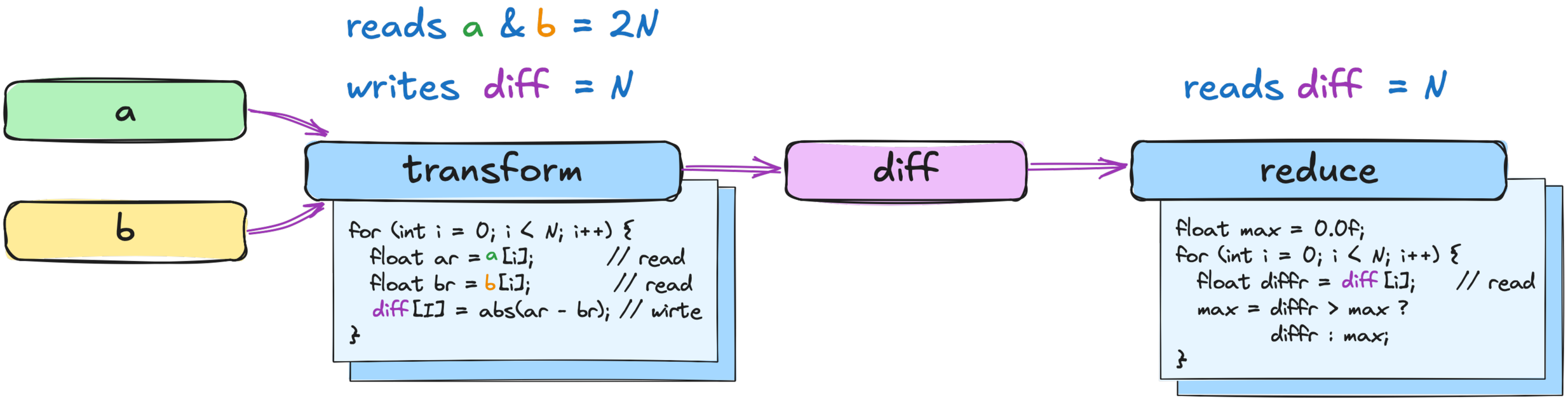
Extending Standard Algorithms

Compute max difference between two vectors

```
thrust::universal_vector<float>
  unnecessarily_materialized_diff(a.size());

// compute abs differences
thrust::transform(
  thrust::device,
  a.begin(), a.end(), // first input sequence
  b.begin(),          // second input sequence
  unnecessarily_materialized_diff.begin(),
  []__host__ __device__(float x, float y) {
    return abs(x - y);
  });

// compute max difference
return thrust::reduce(
  thrust::device,
  unnecessarily_materialized_diff.begin(),
  unnecessarily_materialized_diff.end(),
  0.0f, thrust::maximum<float>{});
```

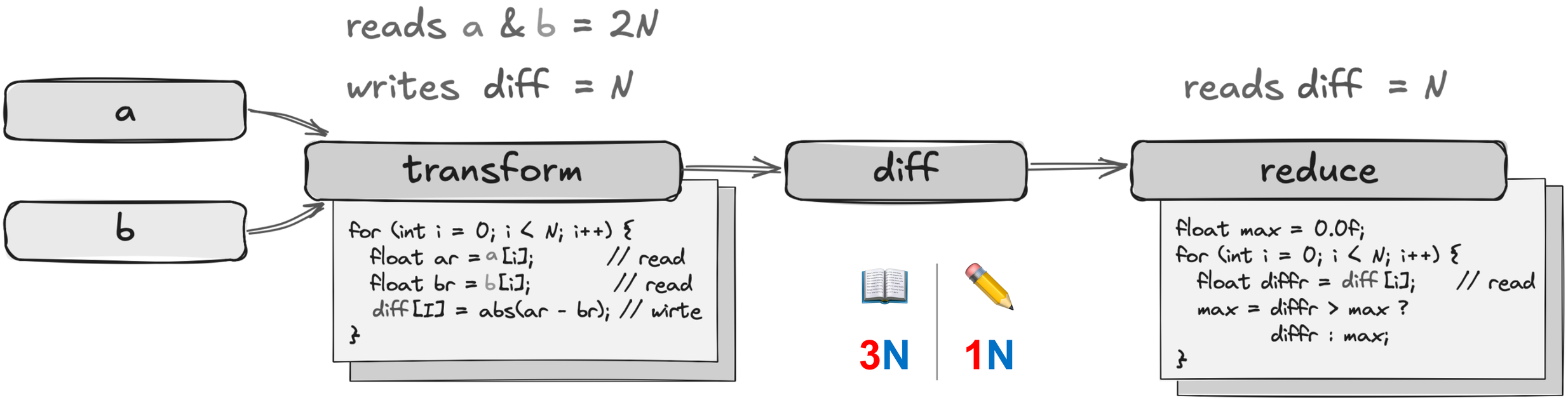


Counting Memory Accesses



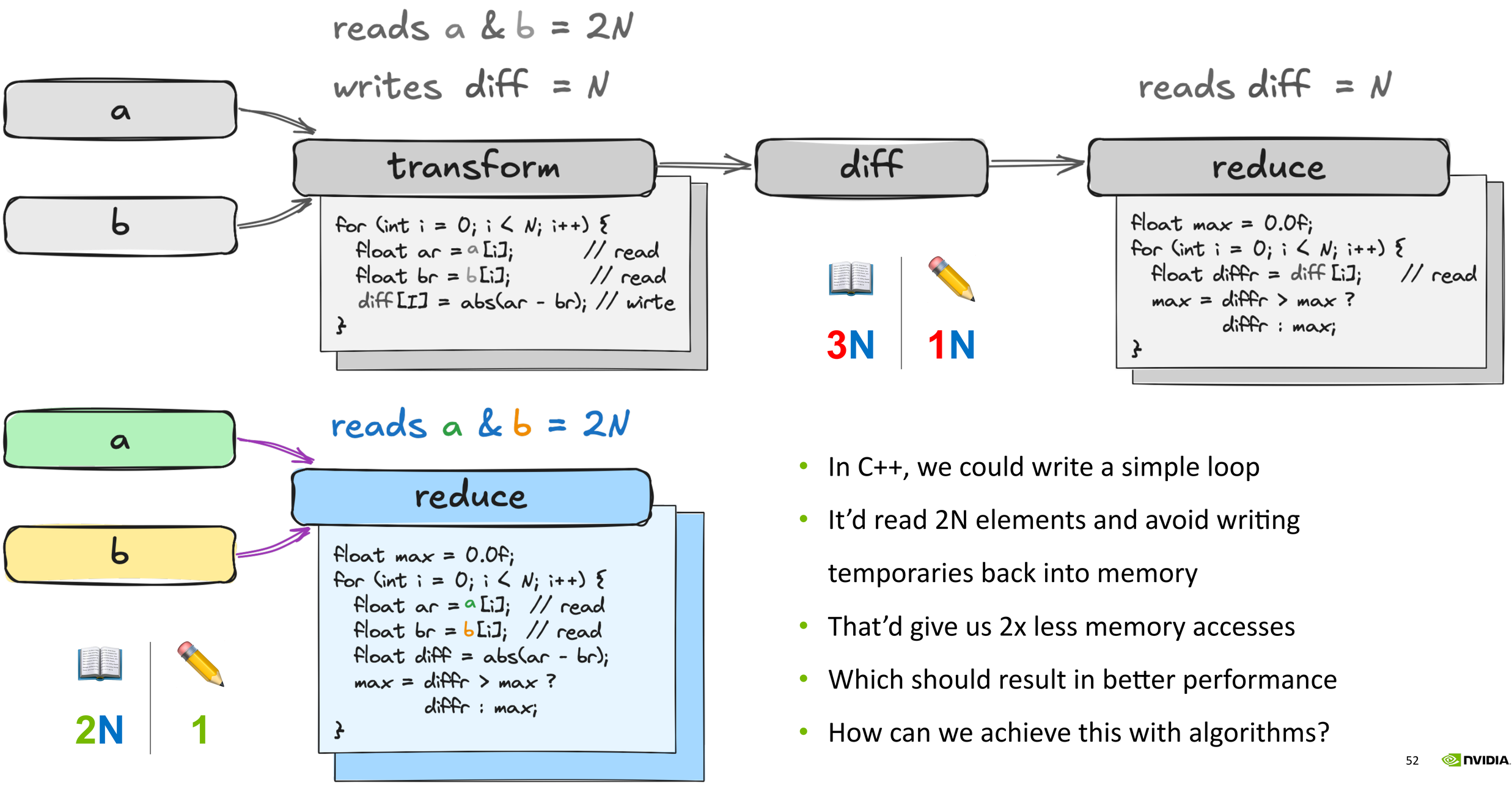
- Algorithm starts with allocation, which is already suboptimal
- After that, transform reads 2N elements and writes N diffs
- After that, reduction has to read N differences and return max
- If you had to implement this algorithm, would you write it this way?

Counting Memory Accesses



- Algorithm starts with allocation, which is already suboptimal
- After that, transform reads 2N elements and writes N diffs
- After that, reduction has to read N differences and return max
- If you had to implement this algorithm, would you write it this way?

Preferred Implementation



- In C++, we could write a simple loop
- It'd read 2N elements and avoid writing temporaries back into memory
- That'd give us 2x less memory accesses
- Which should result in better performance
- How can we achieve this with algorithms?

Pointers

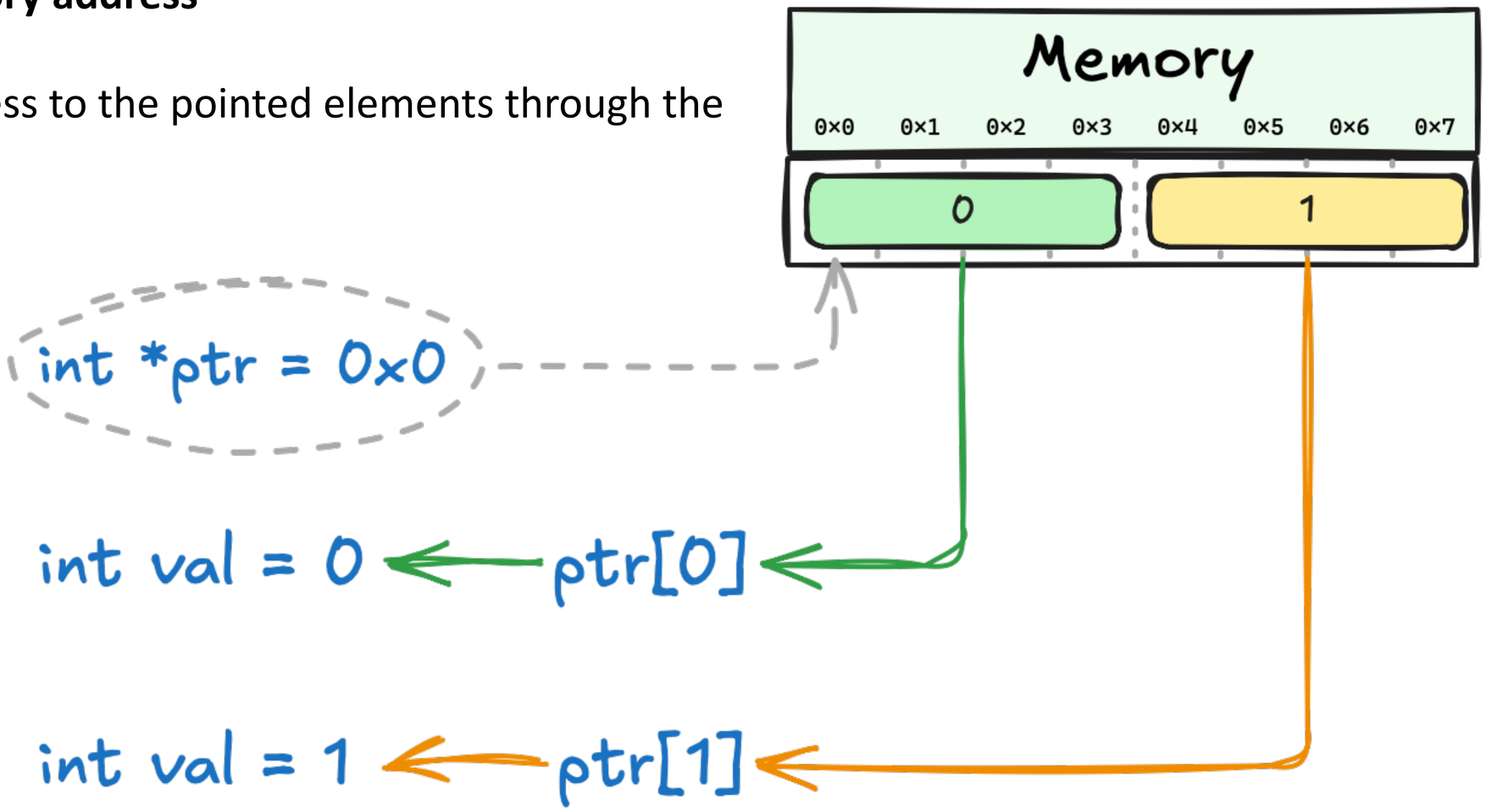
```
std::array<int, 3> a{0, 1, 2};

int *ptr = a.data(); // points to a[0]

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

N

Pointers

A **very** simple mental model for pointers is something that:

- Stores a **physical memory address**
- Enables array-style access to the pointed elements through the subscript operator



Pointers Interface

```
std::printf("pointer[0]: %d\n", ptr[0]);
std::printf("pointer[1]: %d\n", ptr[1]);
```

Subscript operator[]

- Provides array-style element access
- C++ allows overloading of subscript operator[]
- That's how indexing works in vectors vec[42]

```
struct vector
{
  int operator[](int i = 42)
  {
    return *(data() + i);
  }
};
```

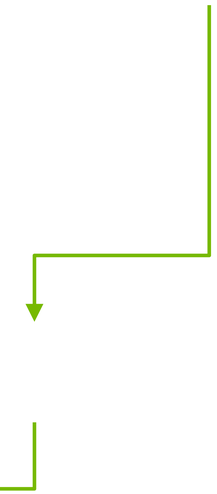
Simple Counting Iterator

- We can overload subscript operator[]
- But instead of accessing physical memory, we can, say, return the incoming index

```
counting_iterator it;

std::printf("it[0]: %d\n", it[42]);
```

Mental Model

```
struct counting_iterator
{

  int operator[](int i = 42)
  {
    return i;
  }

};
```



Simple Counting Iterator

Pointers

- Subscript operator leads to memory accesses

```
std::array<int, 3> a{0, 1, 2};

int *ptr = a.data();

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

N

Iterators

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

```
counting_iterator it;

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

0

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;
  }
};
```

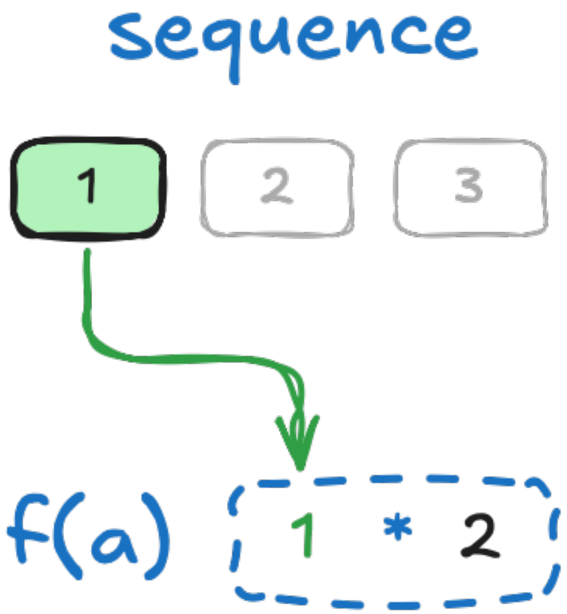
```
counting_iterator it;

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

0

Iterators:

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

Simple Transform Iterator



Mental Model:

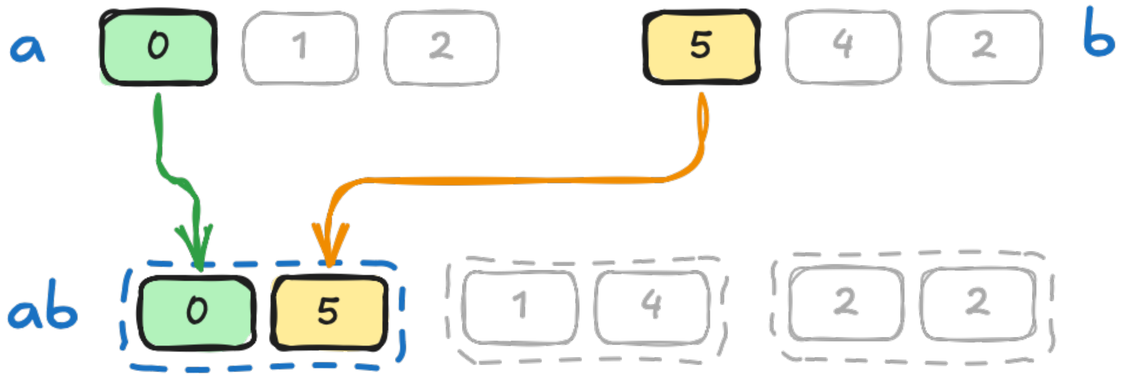
```
struct transform_iterator
{
  int *a;

  int operator[](int i)
  {
    return a[i] * 2;
  }
};
```

Transform iterator applies a function before returning a value

```
std::array<int, 3> a{ 0, 1, 2 };

transform_iterator it{a.data()};

std::printf("it[0]: %d\n", it[0]); // prints 0 std::printf("it[1]: %d\n", it[1]); // prints 2
```

N

Simple Zip Iterator



Mental Model:

```
struct zip_iterator
{
  int *a;
  int *b;

  std::tuple<int, int> operator[](int i)
  {
    return {a[i], b[i]};
  }
};
```

```
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> b{ 5, 4, 2 };

zip_iterator it{a.data(), b.data()};

std::printf("it[0]: (%d, %d)\n", std::get<0>(it[0]), std::get<1>(it[0])); // prints (0, 5)
```

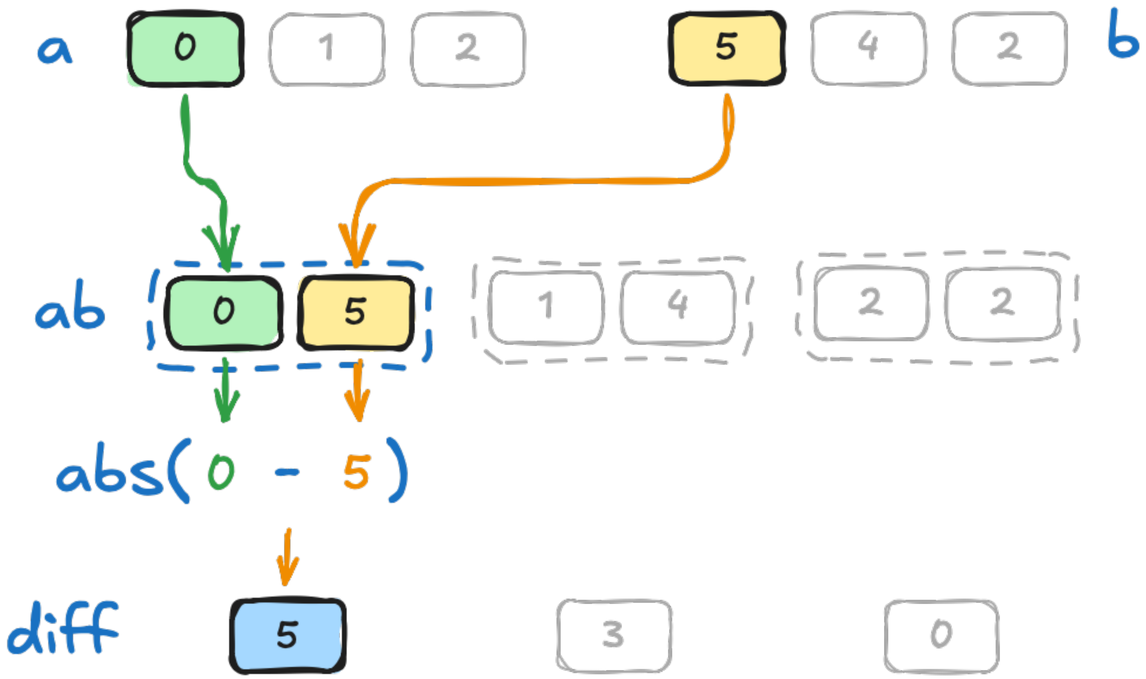
📖

Zip iterator allows you to combine multiple sequences

2N

NVIDIA

Combining Input Iterators



You can nest iterators!

Mental Model:

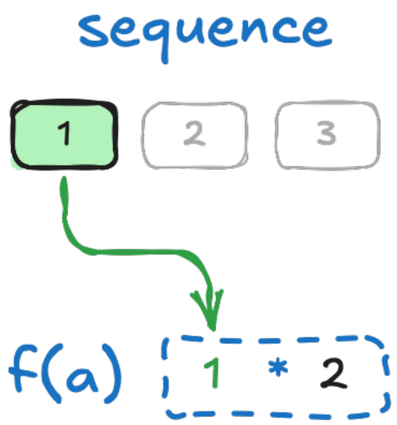
```
struct transform_iterator
{
  zip_iterator zip;

  int operator[](int i)
  {
    auto [a, b] = zip[i];
    return abs(a - b);
  }
};
```

```
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> b{ 5, 4, 2 };

zip_iterator zip{a.data(), b.data()};
transform_iterator it{zip};

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

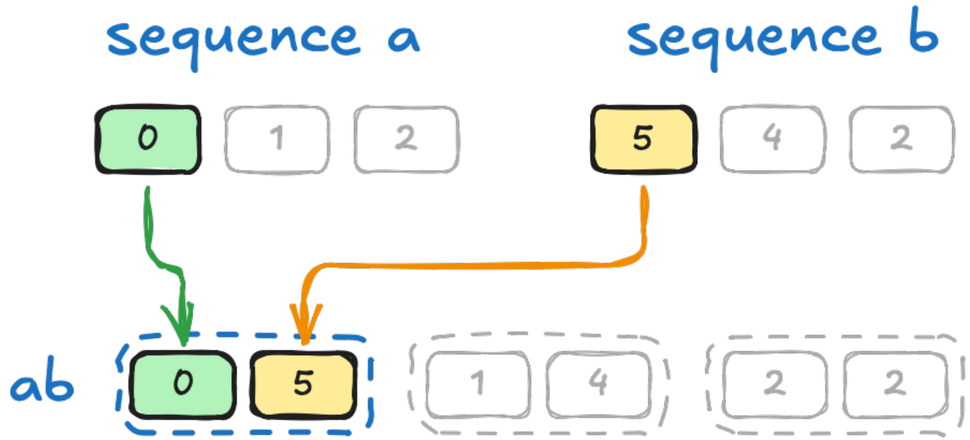
2N

NVIDIA

Thrust Fancy Iterators




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




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

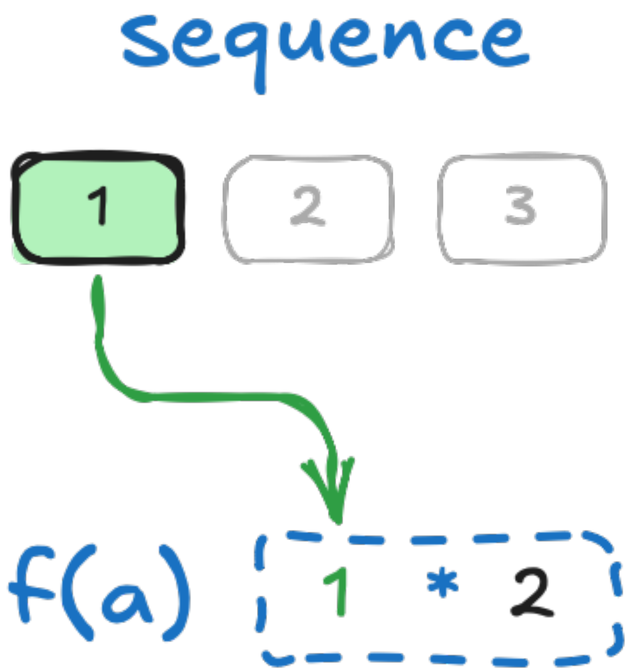

```
thrust::make_counting_iterator(1)
```

Thrust Fancy Iterators

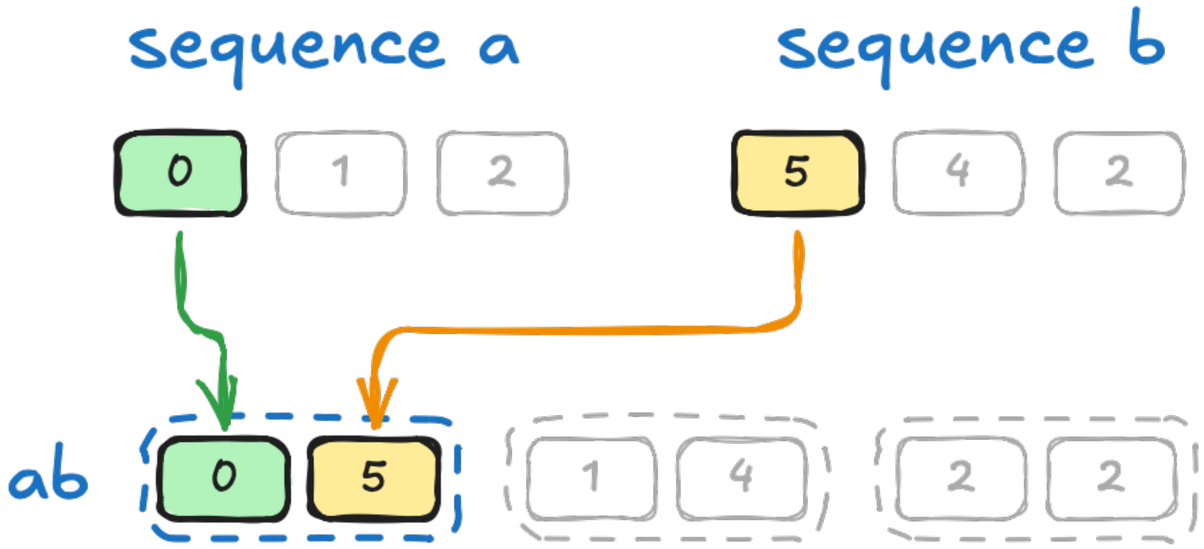

```
auto begin = thrust::make_counting_iterator(1);

auto end = begin + 42;

thrust::for_each(thrust::device, begin, end, print);

// prints: 1 2 3 ... 42
```

Thrust Fancy Iterators

```
thrust::universal_vector<int> vec = {1, 2, 3, ...};
auto begin =
  thrust::make_transform_iterator(

    vec.begin(),

    [] __host__ __device__(int value) {
      return value * 2;
    });

auto end = begin + 42;

thrust::for_each(thrust::device, begin, end, print);

// prints: 2 4 6 8 ... 84
```



Thrust Fancy Iterators



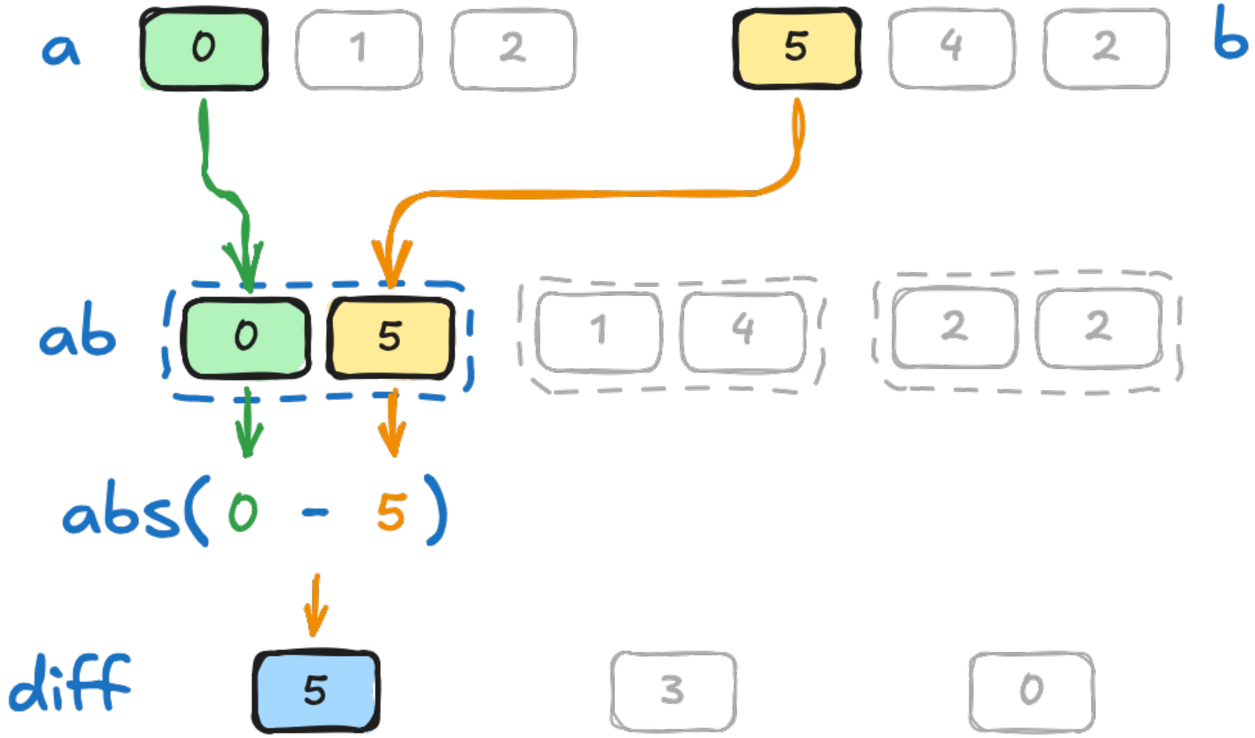

```
thrust::universal_vector<int> a = {0, 1, 2};

thrust::universal_vector<int> b = {5, 4, 2};

auto begin = thrust::make_zip_iterator(a.begin(), b.begin());

auto end = begin + 42;

thrust::for_each(thrust::device, begin, end, print);

// prints: (0, 5) (1, 4) (2, 2)
```

Using Thrust Fancy Iterators

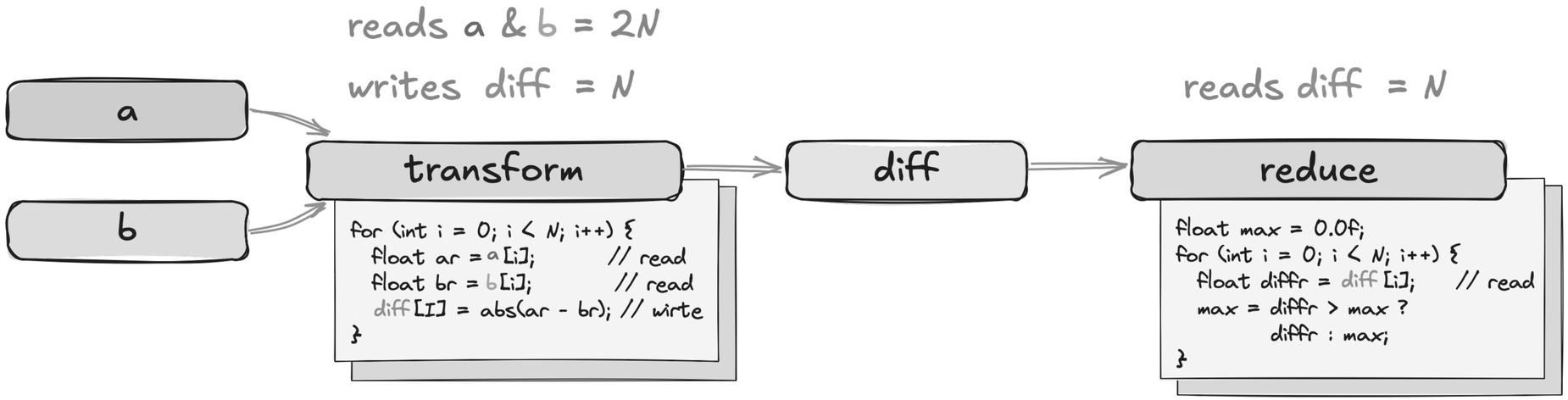
```
auto zip_it = thrust::make_zip_iterator(a.begin(), b.begin());
auto transform_it =
  thrust::make_transform_iterator(zip_it, [] __host__ __device__(thrust::tuple<float, float> t) {
                                    return abs(thrust::get<0>(t) - thrust::get<1>(t));
                                  });
```

Now computation happens upon accessing transform iterator:

- zip_it[i] returns tuple
- transform_it[i] returns diff

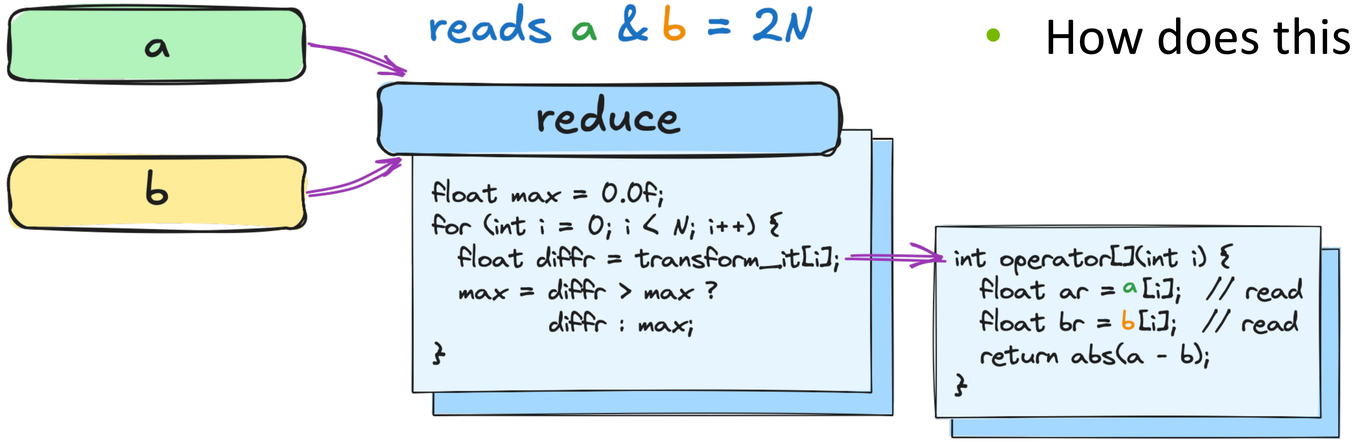


Counting Memory Accesses



Naïve solution

3N | 1N

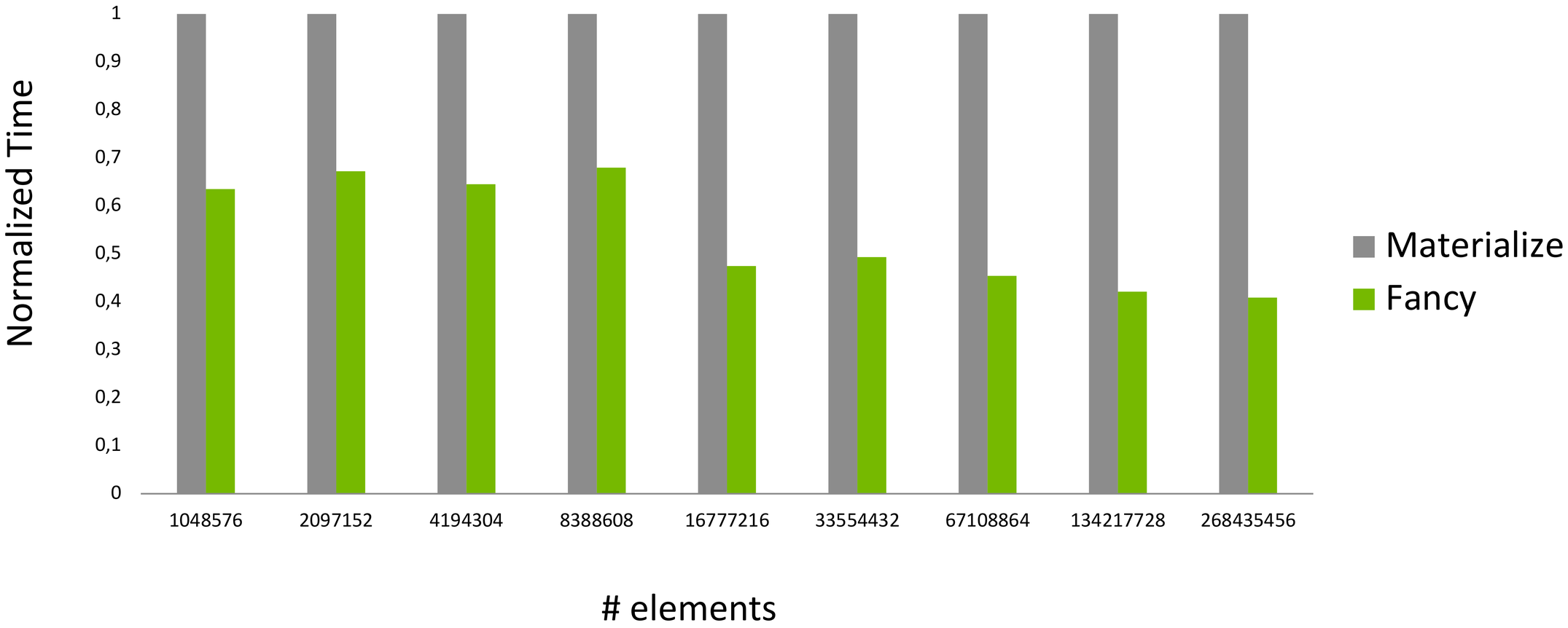


- Version with iterators accesses 2x less memory
- How does this affect performance?

Fancy Iterators

2N | 1

Extending Standard Algorithms



- Version with iterators takes **2x** less time to execute
- Version with iterators needs **30%** less memory

Exercise: Computing Variance

10 minutes

- change the code below so that dereferencing squared_differences returns (x[i] - mean) * (x[i] - mean)

```
float variance(const thrust::universal_vector<float> &x, float mean)
{
  auto squared_differences = ...




  return thrust::reduce(
    thrust::device, squared_differences, squared_differences + x.size()
  ) / x.size();
}
```

2N

N

01.03-Extending-Algorithms/01.03.02-Exercise-Computing-Variance.ipynb

Exercise: Computing Variance

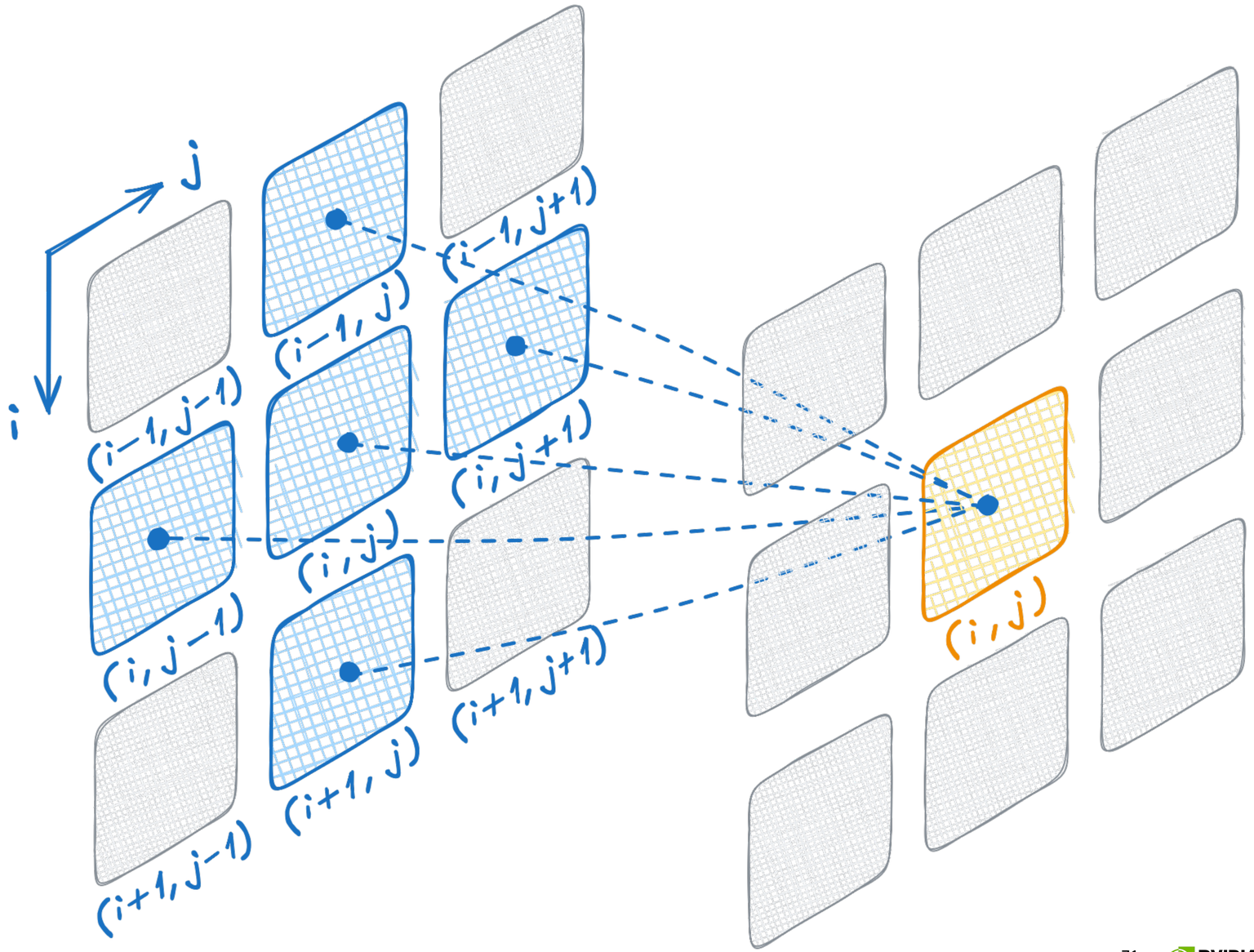
Solution

```
float variance(const thrust::universal_vector<float> &x, float mean)
{
  auto squared_differences = thrust::make_transform_iterator(
    x.begin(), [mean] __host__ __device__(float value) {
      return (value - mean) * (value - mean);
    });

  return thrust::reduce(
    thrust::device, squared_differences, squared_differences + x.size()
  ) / x.size();
}
```


N | 1

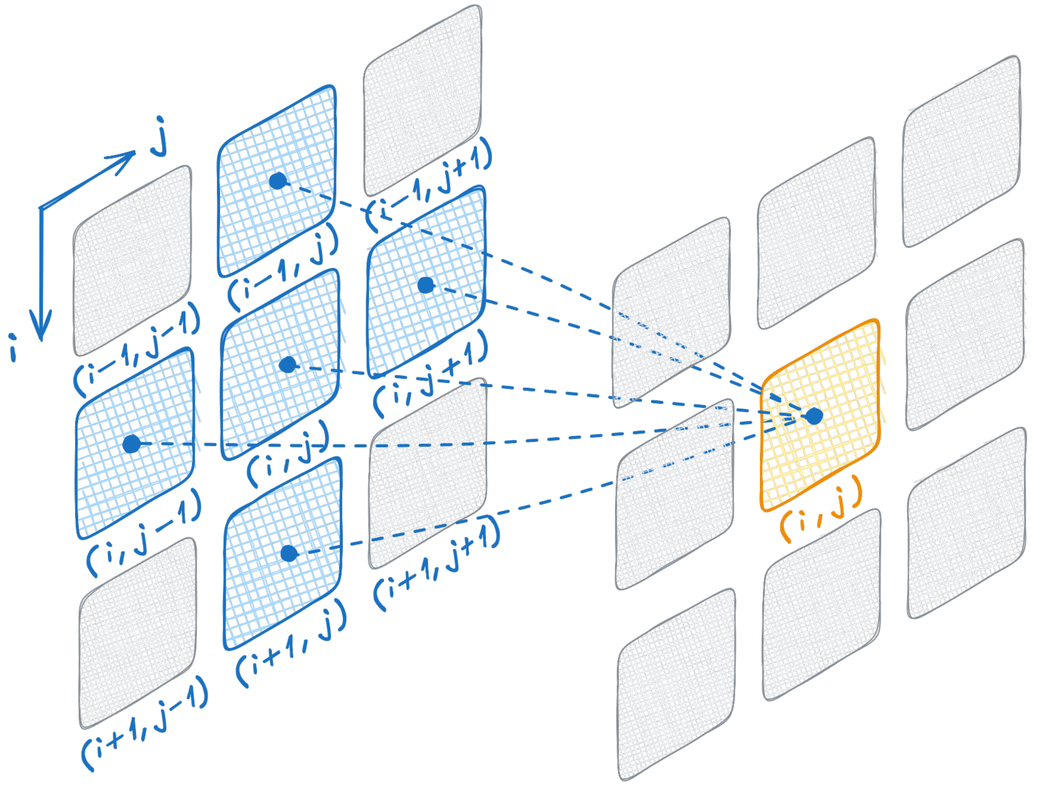
Heat Equation

- Let’s implement an actual heat equation simulator
- Neighbors now affect next state of a given cell
- This pattern is called stencil
- We’ll organize cells into a two-dimensional matrix



Implementing Stencil Pattern

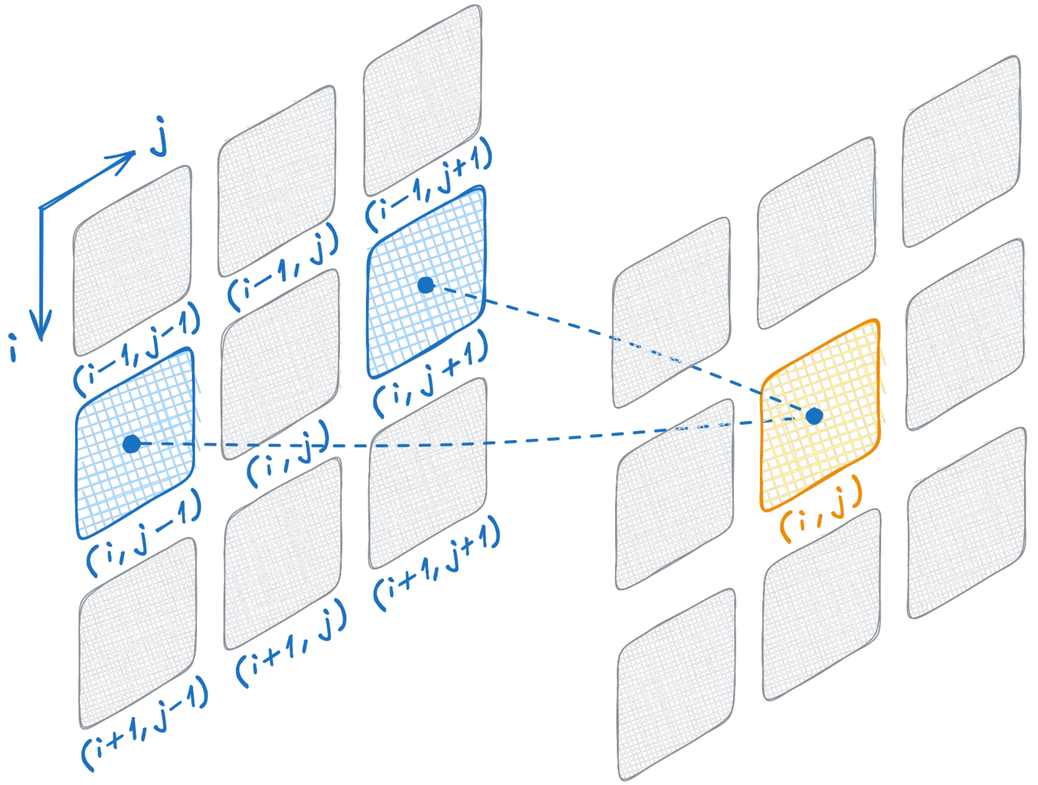
- Don’t worry about the exact formula
- Instead, consider access pattern




```
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);
```

Implementing Stencil Pattern

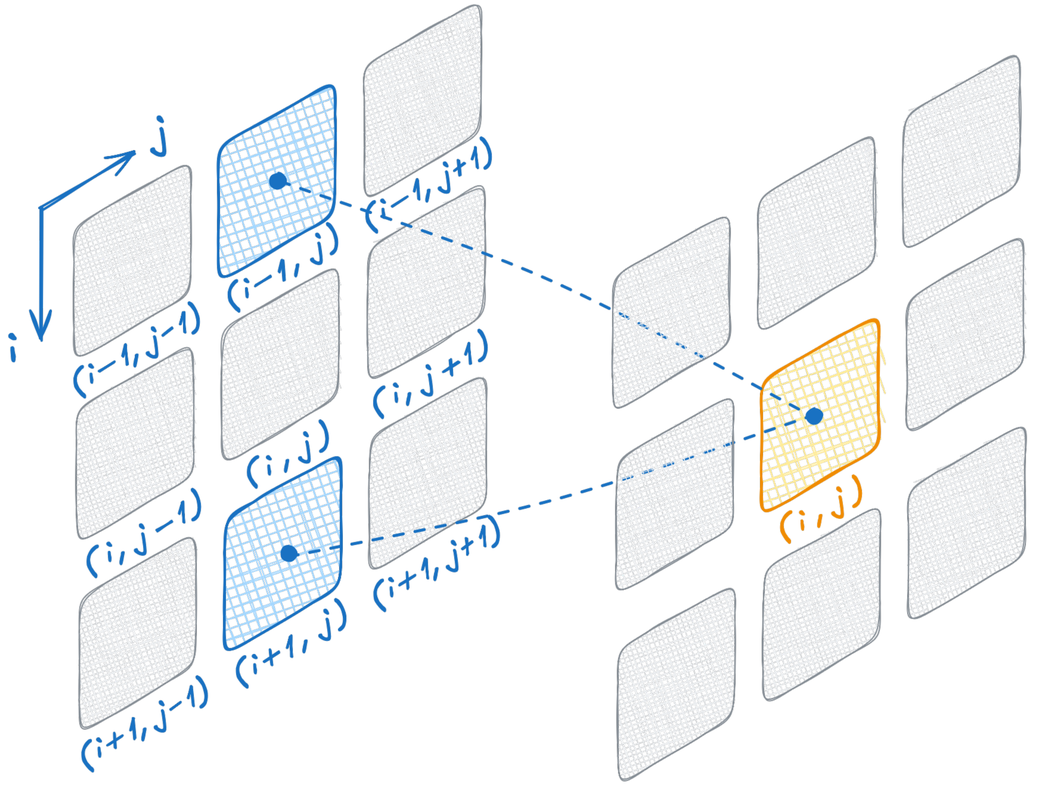
- Don’t worry about the exact formula
- Instead, consider access pattern
  - We need values from left and right




```
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);
```

Implementing Stencil Pattern

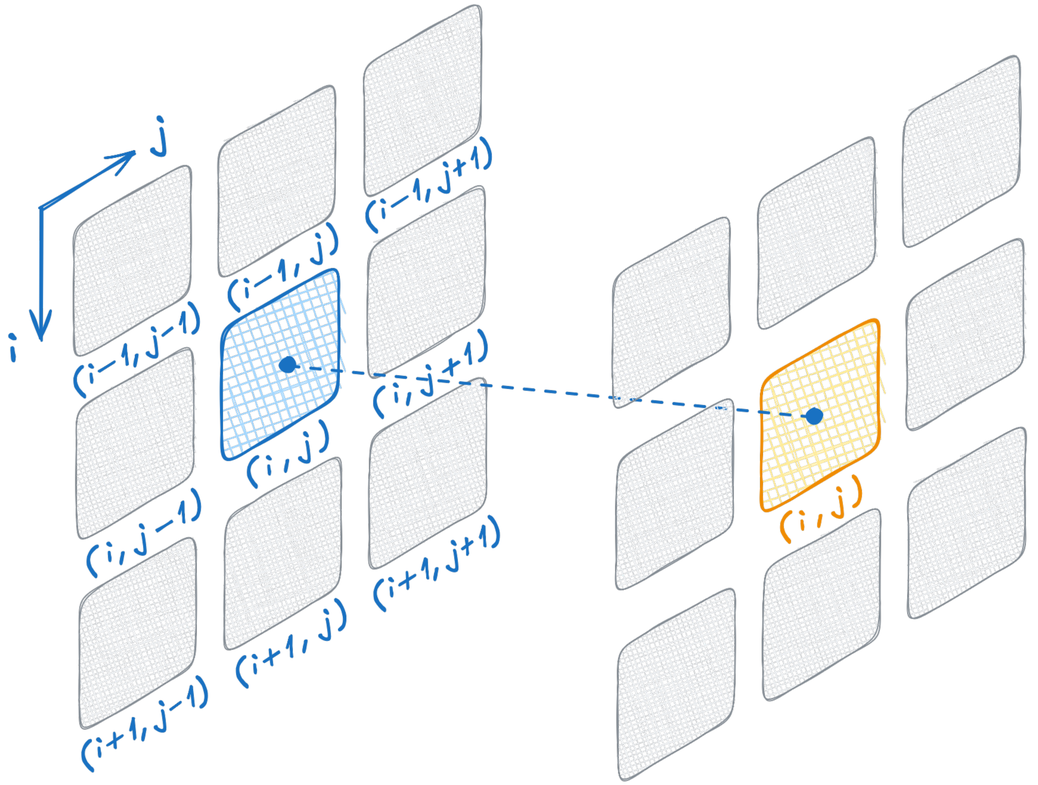
- Don’t worry about the exact formula
- Instead, consider access pattern
  - We need values from left and right
  - We need values from top and bottom




```
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);
```

Implementing Stencil Pattern

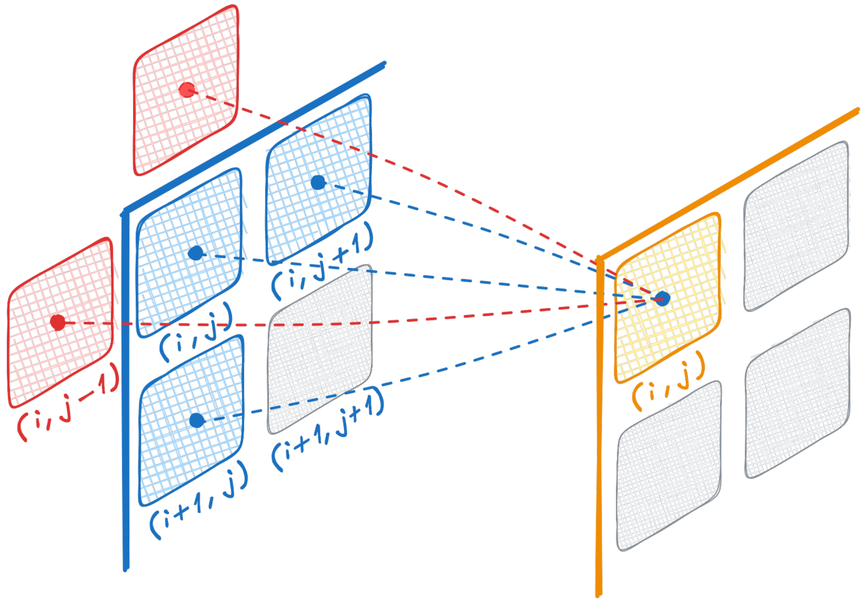
- Don’t worry about the exact formula
- Instead, consider access pattern
  - We need values from left and right
  - We need values from top and bottom
  - We need value of the current cell




```
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);
```

Boundary Conditions

- Our grid is not infinite
- Not all cells have neighbors
- Let's handle boundary conditions
- Cells on boundary will be constant




```
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];
}
```

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

Converting Thrust Pointers to Raw Pointers

```
void simulate(int height, int width,
              const thrust::universal_vector<float> &in,
                    thrust::universal_vector<float> &out)
{
  const float *in_ptr = thrust::raw_pointer_cast(in.data());

  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) {
```

Get raw pointer from Thrust container

```
std::vector<int> vec(42);

int *data = vec.data();
```

std::vector::data()

- returns a *raw pointer*
- Because some interfaces require raw pointers

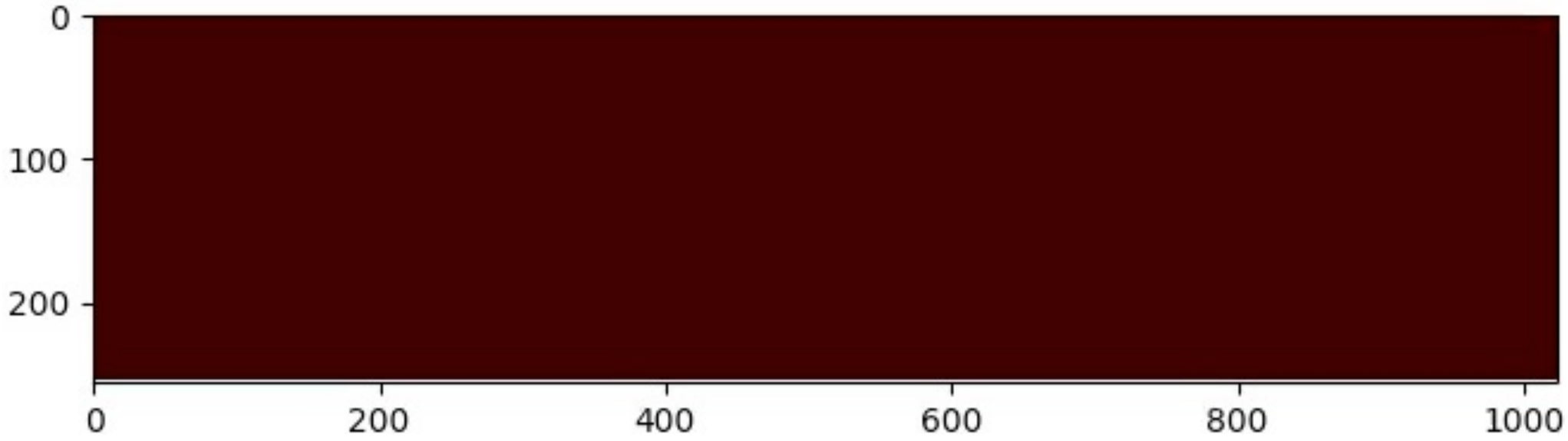
```
thrust::universal_vector<int> vec(42);

thrust::cuda::universal_pointer<int> data = vec.data();

int *ptr = thrust::raw_pointer_cast(vec.data());
```

thrust::universal_vector::data()

- Returns a *typed iterator*
- You can convert it to a raw pointer with thrust::raw_pointer_cast

Implementing Stencil Pattern

Simulation results

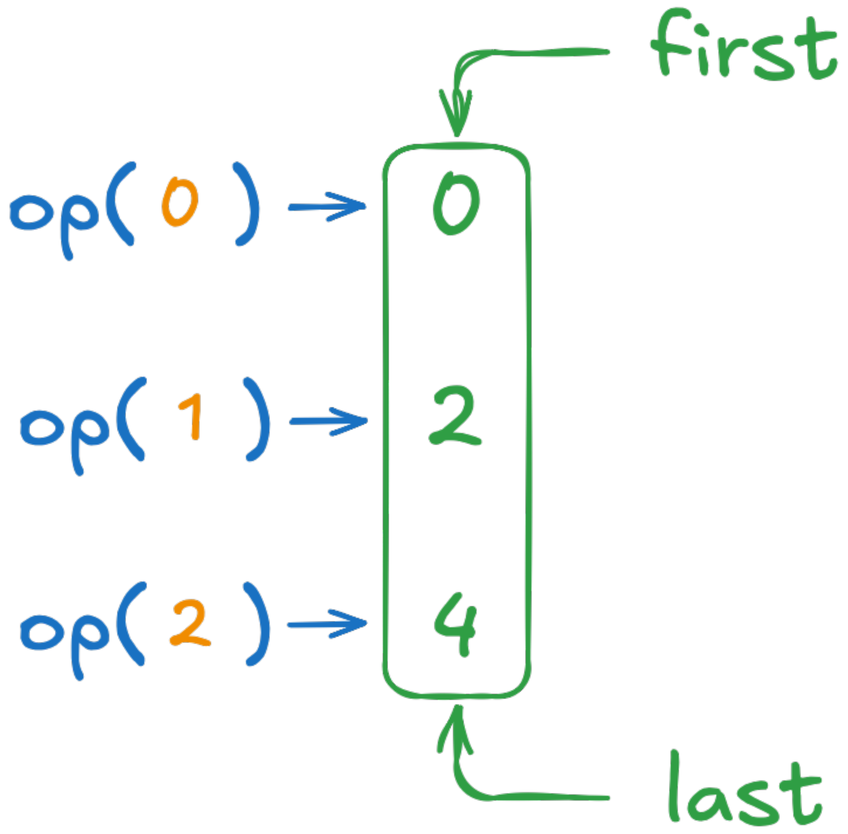


Let’s simulate an empty oven:

- Heating elements on top and bottom
- Cold walls on left and right

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




```
thrust::tabulate( first , last , op )
```

Using Tabulate

```
void simulate(int height, int width,
              const thrust::universal_vector<float> &in,
                    thrust::universal_vector<float> &out)
{
  const float *in_ptr = thrust::raw_pointer_cast(in.data());

  thrust::tabulate(
    thrust::device, out.begin(), out.end(),
    [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];
      }
    });
}
```

Tabulate is equivalent to transform of counting iterator

Code Reuse

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

- 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

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?

Vocabulary Types in libcu++

Compound Types

- cuda::std::pair
- cuda::std::tuple
- ...

Optional and Alternatives

- cuda::std::optional
- cuda::std::variant
- ...

Math

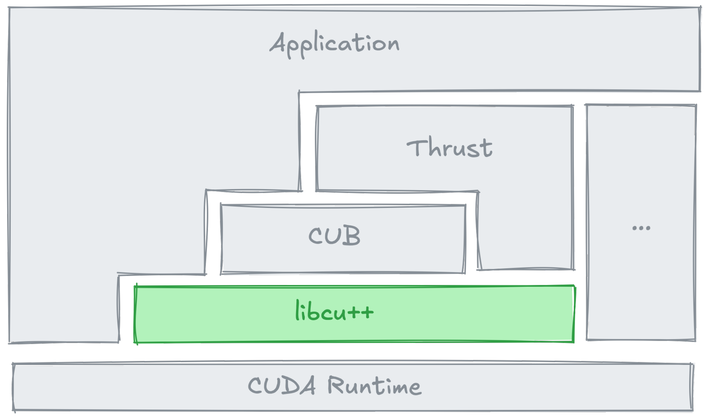
- cuda::std::complex
- cuda::std::mdspan
- ...

Synchronization

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

CUDA Extensions

- cuda::atomic
- cuda::atomic_ref
- cuda::atomic_flag
- ...



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

Vocabulary Types

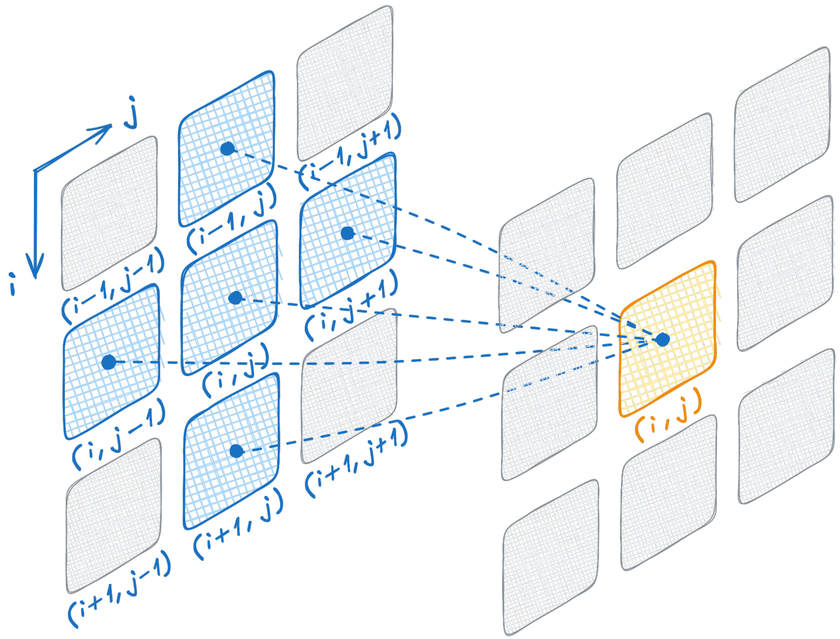
```
__host__ __device__
cuda::std::pair<int, int> row_col(int id, int width)
{
  return cuda::std::make_pair(id / width, id % width);
}
```

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

- Instead of re-implementing vocabulary types
- Just add **cuda::**in front of a standard type and you get a version that works on both **CPU** and **GPU**

Vocabulary Types

- Pair is not the only vocabulary type that can improve our simulator
- Consider how we had to manually flatten 2D coordinates
- This approach is:
  - error prone
  - not productive




```
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];
```

mdspan

Disadvantages of manual linearization become more apparent as number of dimensions increases

```
// inp[b][n][0][nh][d]

int inp_idx =

    (b * N * 3 * NH * d)
+   (n * 3 * NH * d)
+       (0 * NH * d)
+           (nh * d)
+                d;

float q = inp[inp_idx]);
```

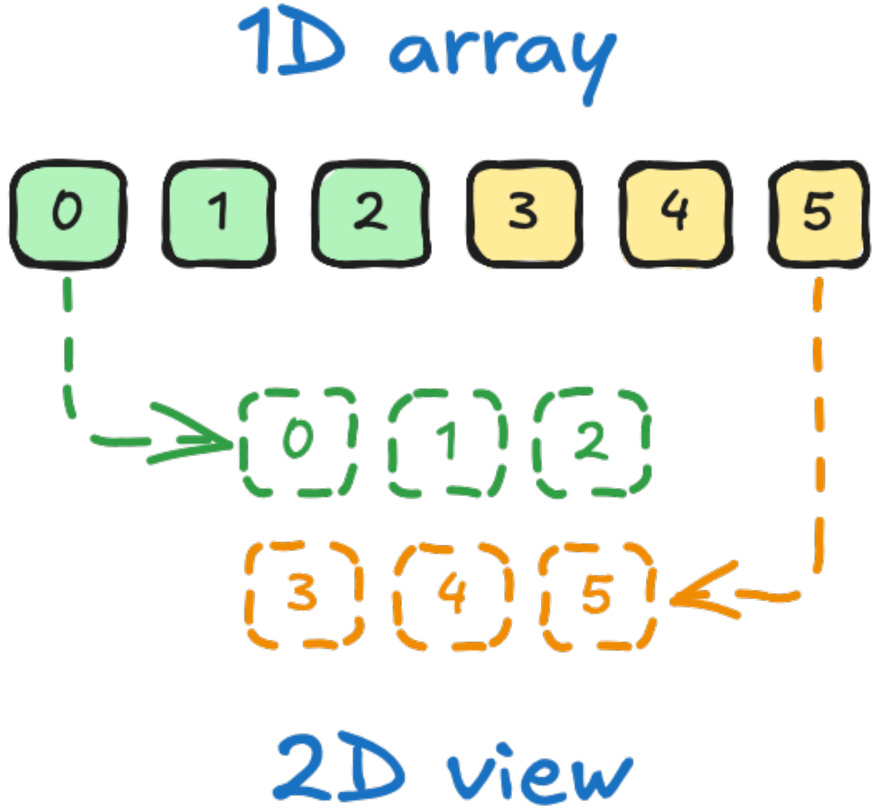
```
float q = inp(b, n, 0, nh, d)
```

- There’s a standard type that can turn comment on the left into actual code at no performance cost

mdspan

cuda::std::mdspan

- a multidimensional view of a sequence, that
- helps write clean code, and
- helps avoid bugs
  - in linearization of multidimensional indices
  - by coupling dimensions with the view itself



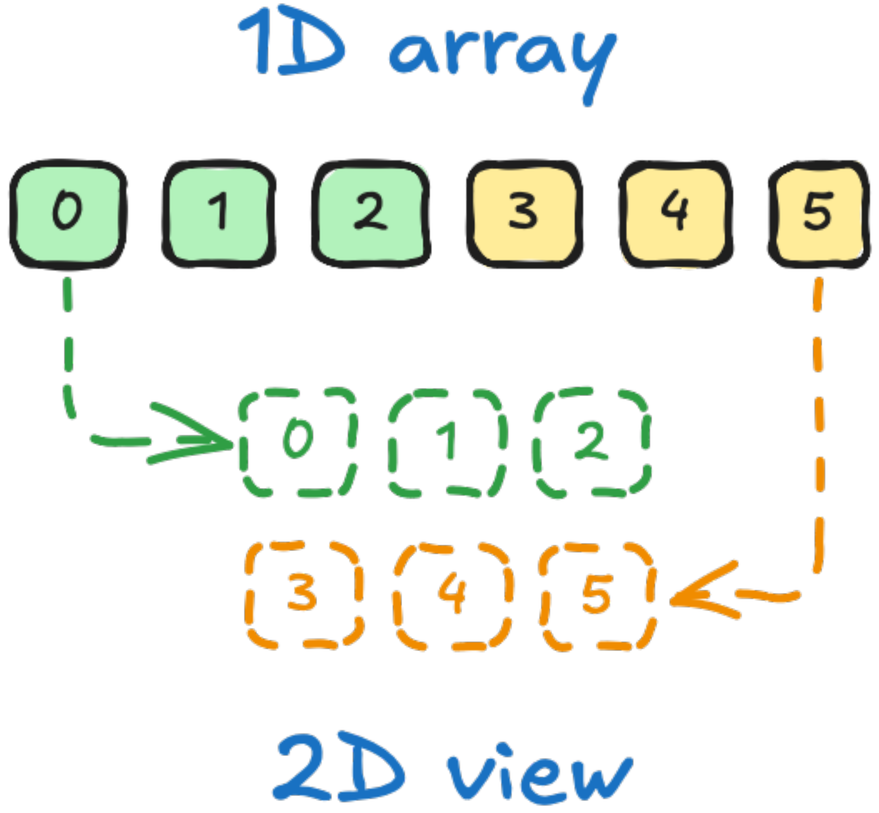
mdspan

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

Let’s start with a 1D array that we’d like to ***view*** as 2D matrix

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

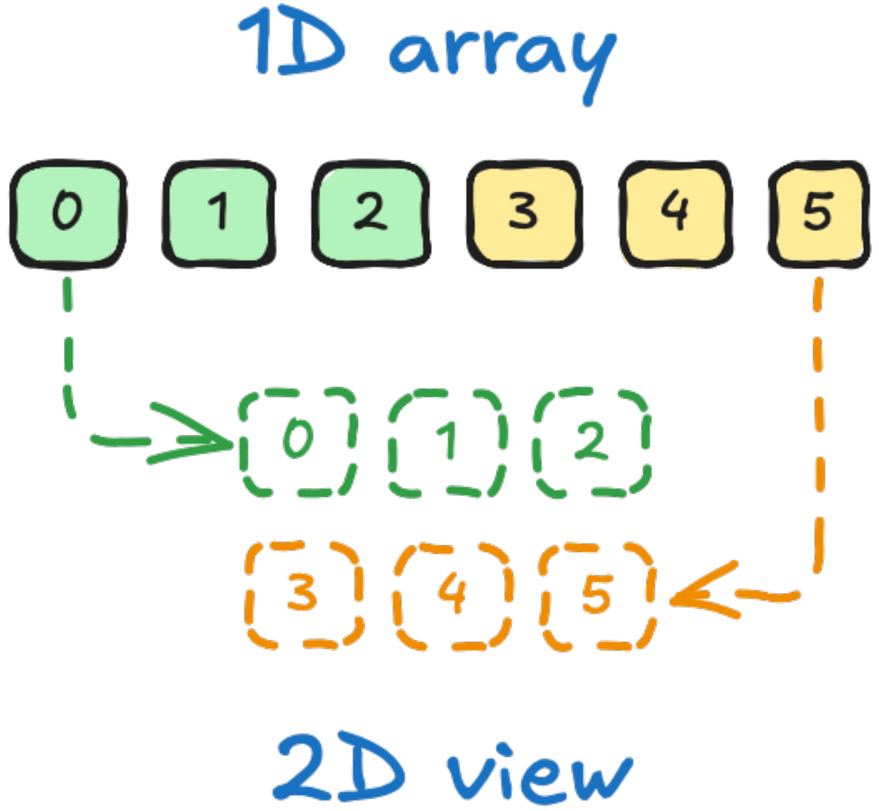
mdspan

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

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

std::printf("md(0, 0) = %d\n", md(0, 0)); // 0
std::printf("md(1, 2) = %d\n", md(1, 2)); // 5
```

You can then access the underlying sequence with * operator() by providing:

- Row index
- Column index

* in the case of 2D matrix



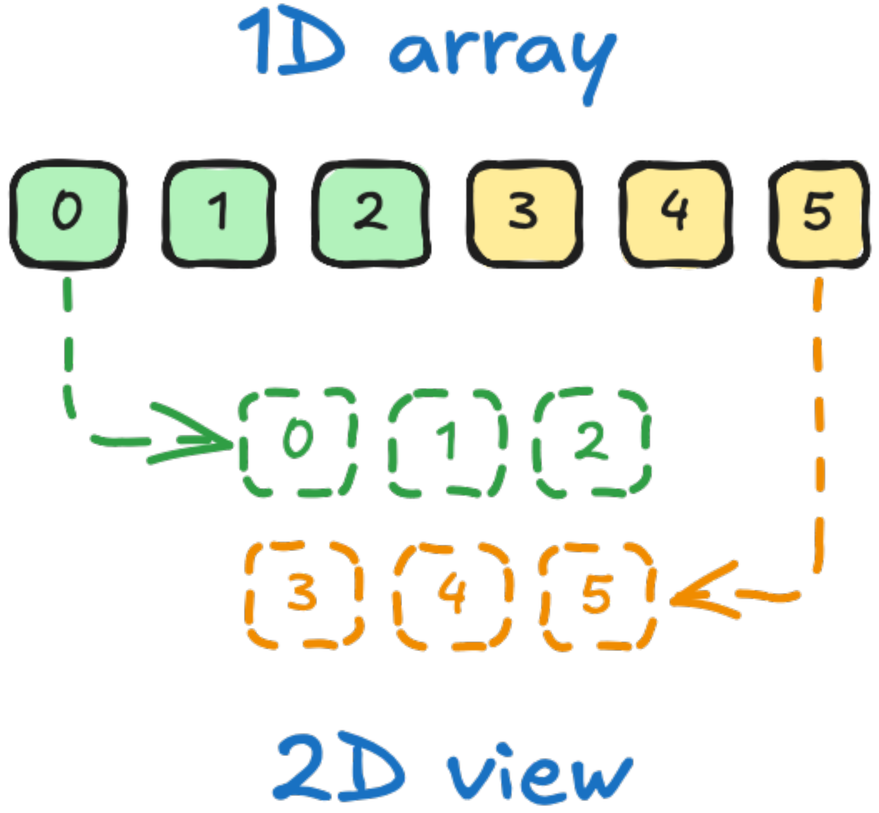
mdspan

cuda::std::mdspan::size()

- Returns product of extents

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

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

std::printf("md(0, 0) = %d\n", md(0, 0)); // 0
std::printf("md(1, 2) = %d\n", md(1, 2)); // 5

std::printf("size = %zu\n", md.size());      // 6
```

You can query total number of elements with `.size()`

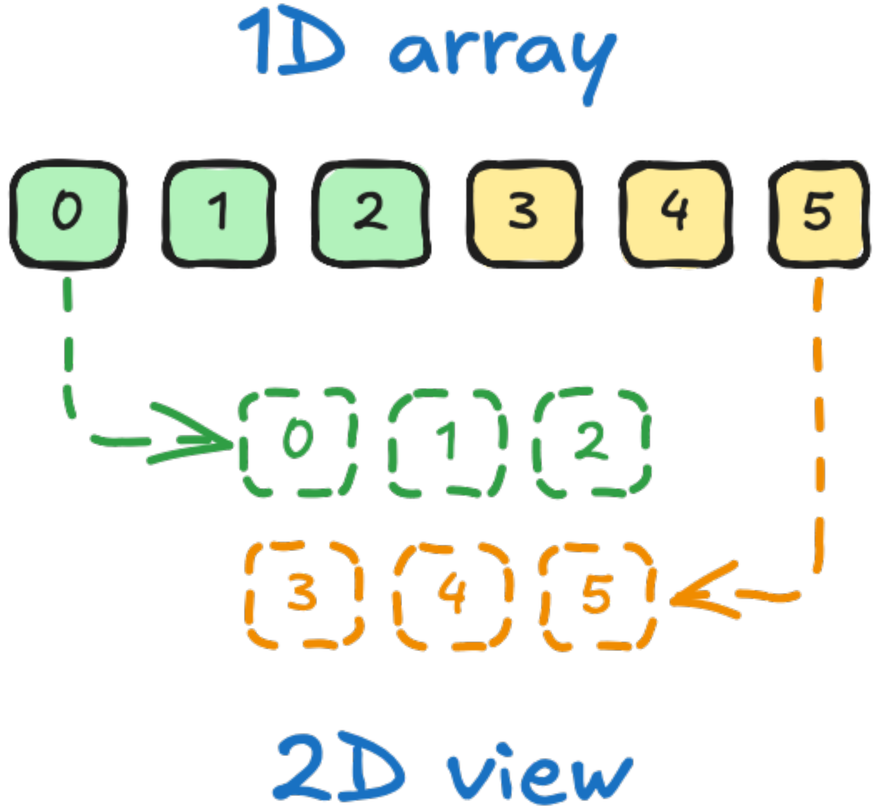


mdspan

cuda::std::mdspan::extent(r)

- Returns extent at rank index r

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

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

std::printf("md(0, 0) = %d\n", md(0, 0)); // 0
std::printf("md(1, 2) = %d\n", md(1, 2)); // 5

std::printf("size = %zu\n", md.size());      // 6
std::printf("height = %zu\n", md.extent(0)); // 2
std::printf("width = %zu\n", md.extent(1));  // 3
```



You can also query:

- The height of the matrix with .extent(0)
- The width of the matrix with.extent(1)

Exercise: mdspan

6 minutes

- Use cuda::std::mdspan instead of raw pointers

```
const float *in_ptr = thrust::raw_pointer_cast(in.data());

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

01.04-Vocabulary-Types/01.04.02-Exercise-mdspan.ipynb

Exercise: mdspan

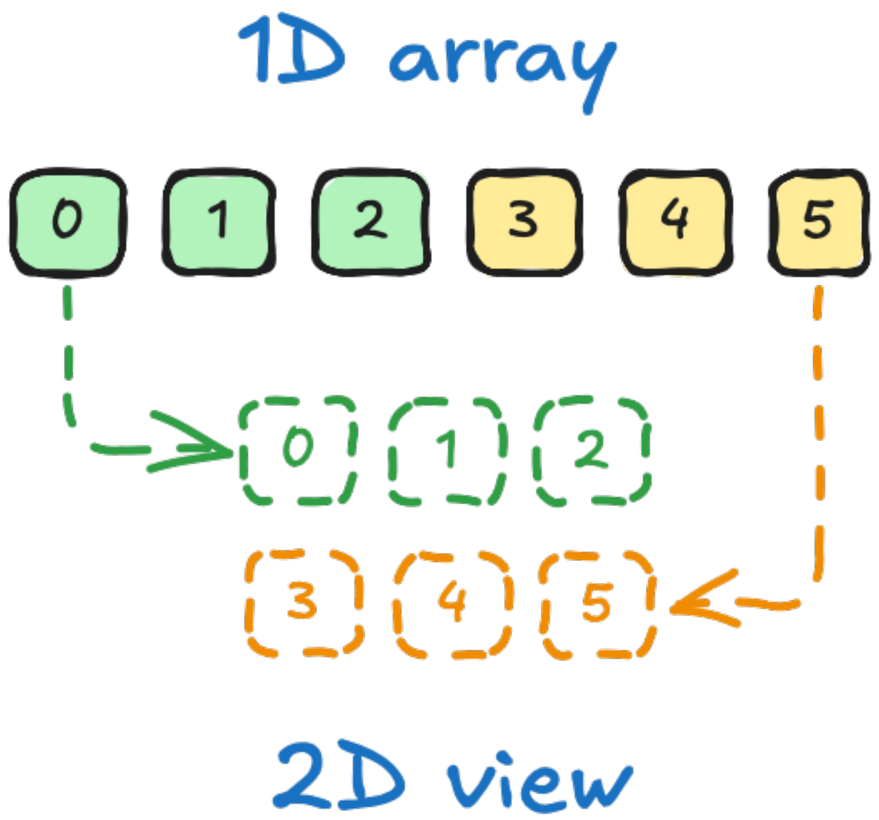
6 minutes

- Use cuda::std::mdspan instead of raw pointers

cuda::std::mdspan::extent(r)

- Returns extent at rank index r

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

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

std::printf("md(0, 0) = %d\n", md(0, 0)); // 0
std::printf("md(1, 2) = %d\n", md(1, 2)); // 5

std::printf("size = %zu\n", md.size());      // 6
std::printf("height = %zu\n", md.extent(0)); // 2
std::printf("width = %zu\n", md.extent(1));  // 3
```



01.04-Vocabulary-Types/01.04.02-Exercise-mdspan.ipynb

Exercise: mdspan

Solution

```
cuda::std::mdspan temp_in(thrust::raw_pointer_cast(in.data()), height, width);

thrust::tabulate(
  thrust::device, out.begin(), out.end(),
  [temp_in] __host__ __device__(int id) {
    int column = id % temp_in.extent(1);
    int row    = id / temp_in.extent(1);

    if (row > 0 && column > 0 && row < temp_in.extent(0) - 1 && column < temp_in.extent(1) - 1) {
      float d2tdx2 = temp_in(row, column - 1) - 2 * temp_in(row, column) + temp_in(row, column + 1);
      float d2tdy2 = temp_in(row - 1, column) - 2 * temp_in(row, column) + temp_in(row + 1, column);

      return temp_in(row, column) + 0.2f * (d2tdx2 + d2tdy2);
    }
    else {
      return temp_in(row, column);
    }
  });
}
```

Best Practice: Give Domain-Specific Names

```
using temperature_grid_f = cuda::std::mdspan<float, cuda::std::dextents<int, 2>>;

using temperature_grid_d = cuda::std::mdspan<double, cuda::std::dextents<int, 2>>;


void simulate(int height, int width,
              thrust::universal_vector<float> &in,
              thrust::universal_vector<float> &out)
{
  temperature_grid_f temp(thrust::raw_pointer_cast(in.data()), height, width);


  thrust::tabulate(thrust::device, out.begin(), out.end(), [=] __host__ __device__(int
```

- Actual cuda::std::mdspan type is more involving because it allows you to pass static information about some extents
- It's recommended to give your cuda::std::mdspan a domain-specific type aliases

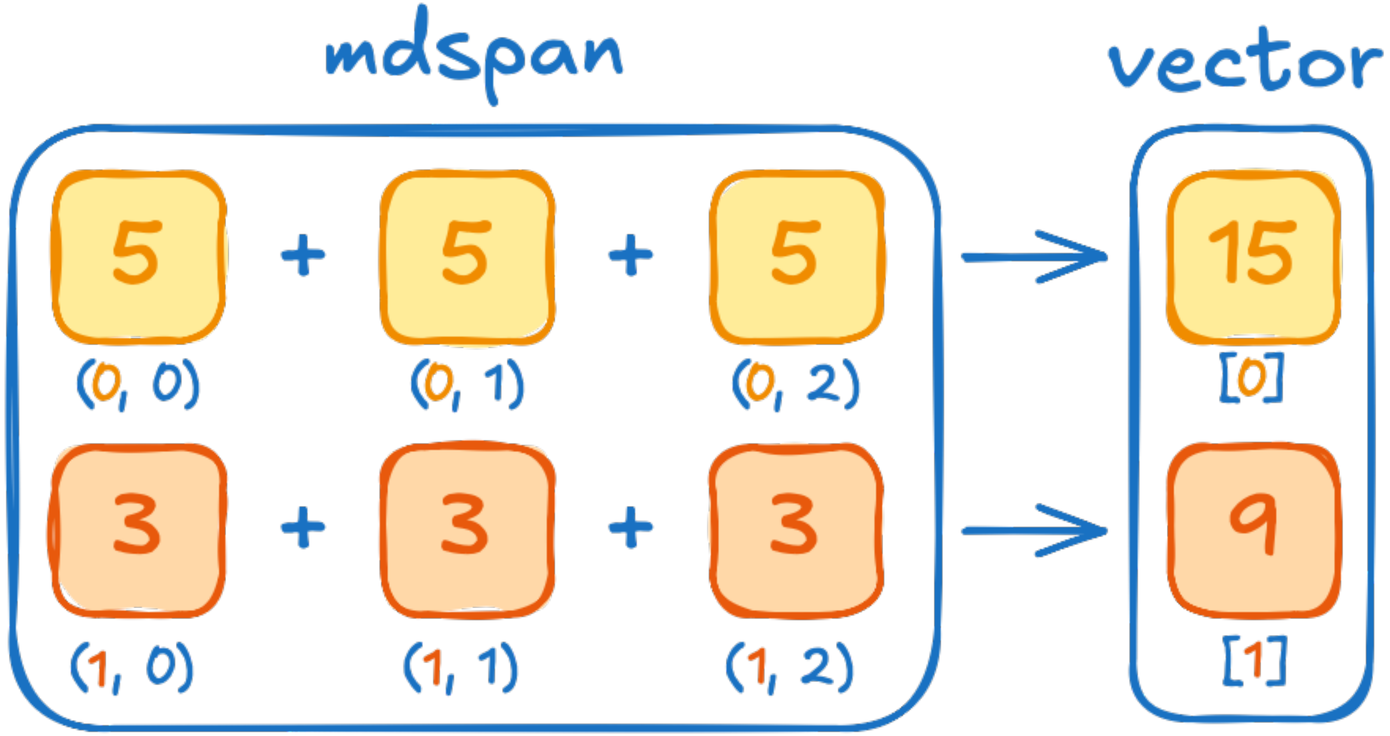
Best Practice: DRY

```
__host__ __device__
float compute(int cell_id, temperature_grid_f temp)
{
  int height = temp.extent(0);
  int width  = temp.extent(1);
  int column = cell_id % width;
```

- Extract commonly used functions, just like you would in standard C++

```
  }
  else
  {
    return temp(row, column);
  }
}

temperature_grid_f temp(thrust::raw_pointer_cast(in.data()), height, width);

thrust::tabulate(thrust::device, out.begin(), out.end(), [=]__host__ __device__(int id) {
  return compute(id, in);
}
```

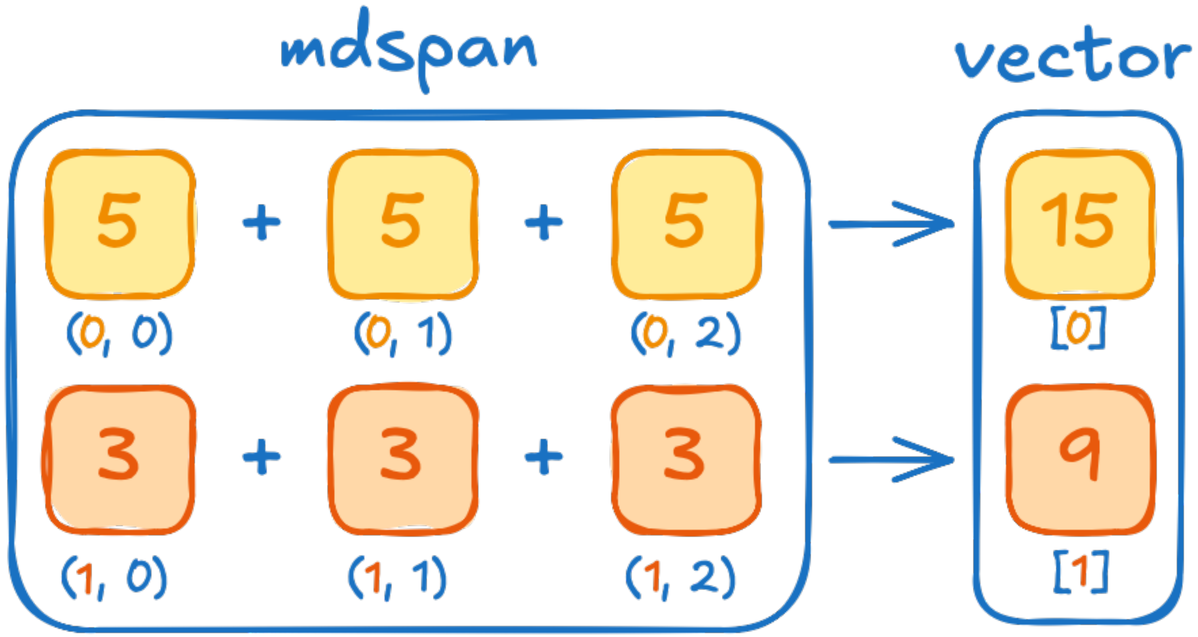
Serial vs Parallel



Let’s compute total temperature in each row

Serial vs Parallel

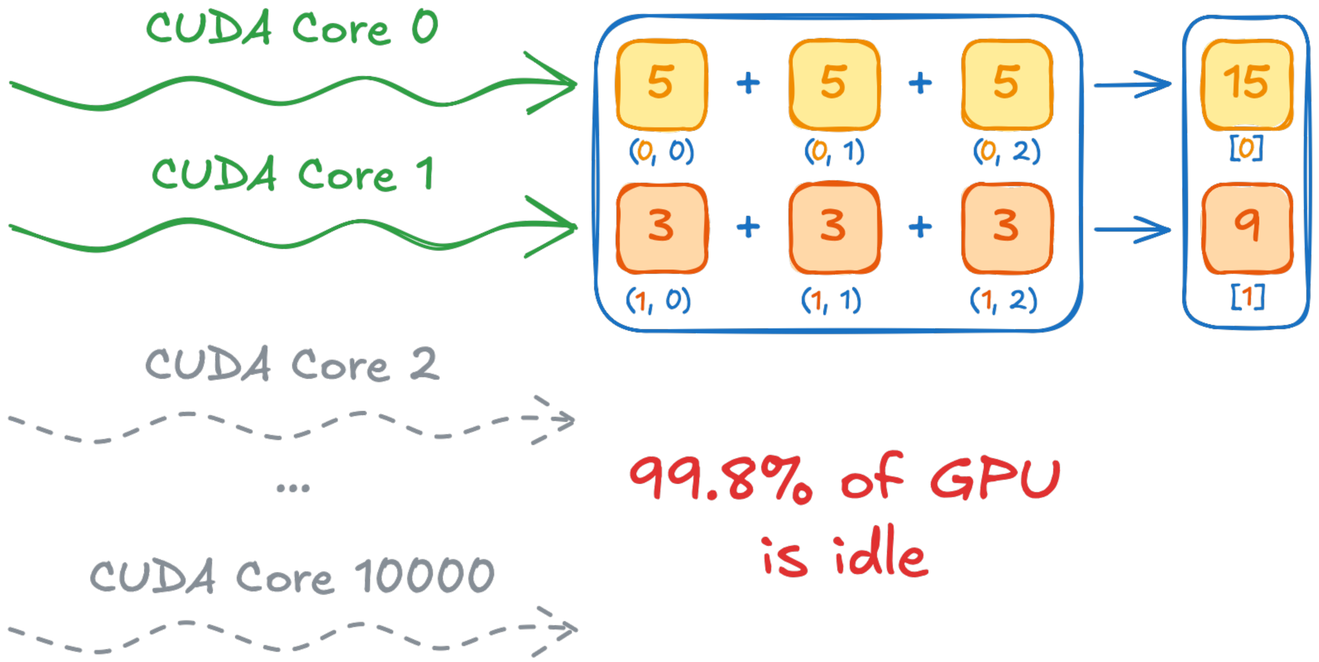
```
thrust::tabulate(
  thrust::device, sums.begin(), sums.end(),
  [=]__host__ __device__(int row_id) {
    float sum = 0;
    for (int col = 0; col < width; col++) {
      sum += temp(row, col);
    }
    return sum;
  });
```



- We can use tabulate again, and iterate each row in the unary operator
- But would that be a good solution?

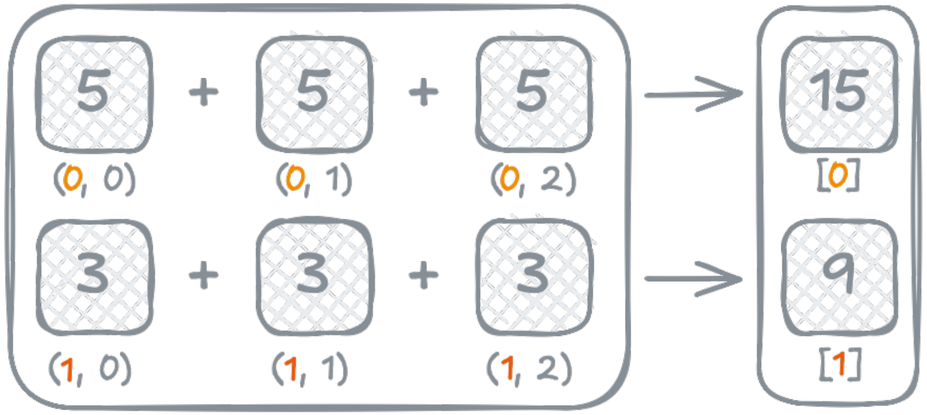
Serial vs Parallel

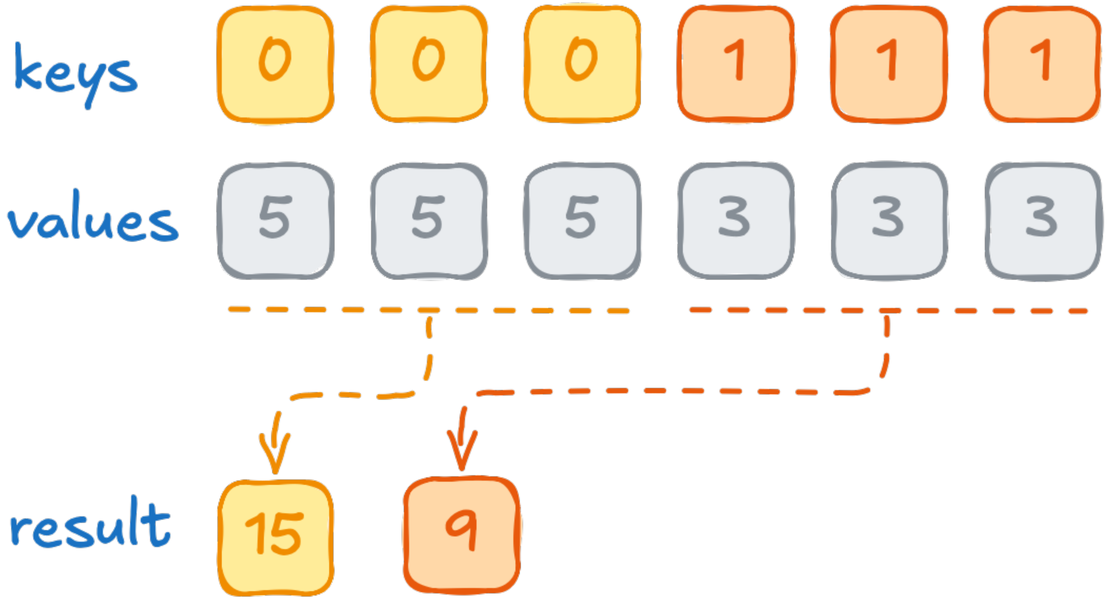
```
thrust::tabulate(
  thrust::device, sums.begin(), sums.end(),
  [=]__host__ __device__(int row_id) {
    float sum = 0;
    for (int col = 0; col < width; col++) {
      sum += temp(row, col);
    }
    return sum;
  });
```



- Code on GPU does not magically become parallel
- This means that the code in lambda function is serial
- This prevents GPU from doing what it's good at – massive parallelism

Serial vs Parallel





- Observe how all values we want to reduce have common row
- Another non-standard Thrust algorithm is a generalization of reduction called: thrust::reduce_by_key
- It reduces values in groups of consecutive keys that are equal
- This algorithm provides better parallelism than our solution, because it'd associate many threads with each row

Serial vs Parallel

```
thrust::universal_vector<float> row_ids(height * width);
thrust::tabulate(
  row_ids.begin(), row_ids.end(),
  [=]__host__ __device__(int i) { return i / width; });
```

- We can materialize keys using tabulate
- Key is essentially a cell index divided by the number of columns

Serial vs Parallel

```
thrust::universal_vector<float> row_ids(height * width);
thrust::tabulate(
  row_ids.begin(), row_ids.end(),
  [=]__host__ __device__(int i) { return i / width; });

thrust::universal_vector<float> sums(height);
thrust::reduce_by_key(
  thrust::device,
  row_ids.begin(), row_ids.end(),   // input keys
  temp.begin(),                     // input values
  thrust::make_discard_iterator(),  // output keys
  sums.begin());                    // output values
```

- Apart from aggregates, reduce by key provides keys as output
- We are not interested in them
- We can save bandwidth using discard iterator

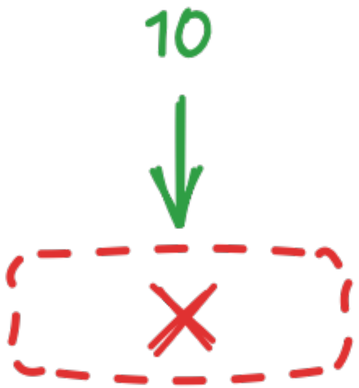
Serial vs Parallel

```
thrust::universal_vector<float> row_ids(height * width);
thrust::tabulate(
  row_ids.begin(), row_ids.end(),
  [=]__host__ __device__(int i) { return i / width; });

thrust::universal_vector<float> sums(height);
thrust::reduce_by_key(
  thrust::device,
  row_ids.begin(), row_ids.end(),   // input keys
  temp.begin(),                     // input values
  thrust::make_discard_iterator(),  // output keys
  sums.begin());                    // output values
```


2N | **2N**

- Apart from aggregates, reduce by key provides keys as output
- We are not interested in them
- We can save bandwidth using discard iterator

Discard Iterator



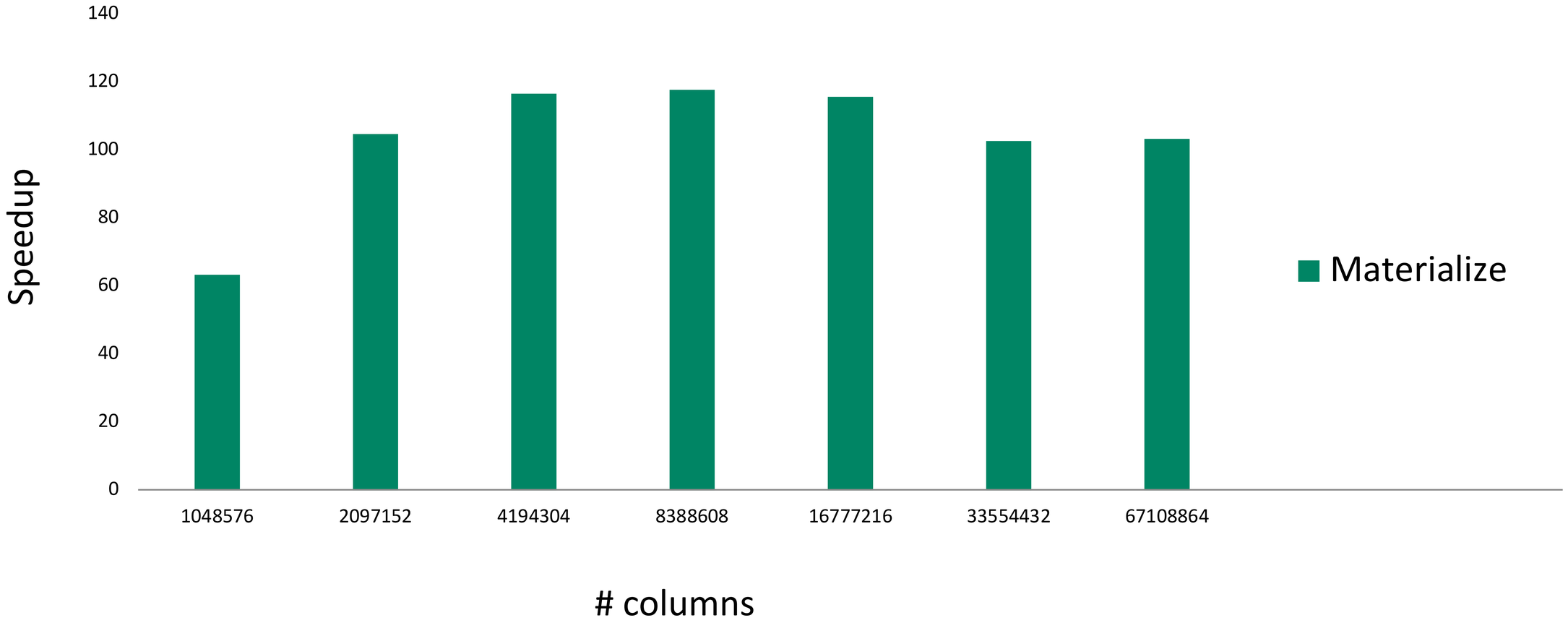
thrust::make_discard_iterator()

- We can even discard the value
- This iterator is helpful when you don't need some of the algorithm's outputs

0

```
struct wrapper
{
  void operator=(int value)
  {
    // discard value
  }
};

struct discard_iterator
{
  wrapper operator[](int i) { return {}; }
};

int main()
{
  discard_iterator it{};

  it[0] = 10;
  it[1] = 20;
}
```

tabulate vs reduce_by_key



- Reduce by key is **100x** faster

Exercise: Segmented Sum Optimization

10 minutes

- Use counting and transform iterators to generate row indices without materializing them in memory

```
auto row_ids_begin = // ... ;



auto row_ids_end = row_ids_begin + temp.size();

thrust::universal_vector<float> sums(height);

thrust::reduce_by_key(thrust::device,
                      row_ids_begin, row_ids_end,
                      temp.begin(), thrust::make_discard_iterator(),
                      sums.begin());
```


2N | 2N

01.05-Serial-vs-Parallel/01.05.02-Exercise-Segmented-Sum-Optimization.ipynb

Exercise: Segmented Sum Optimization

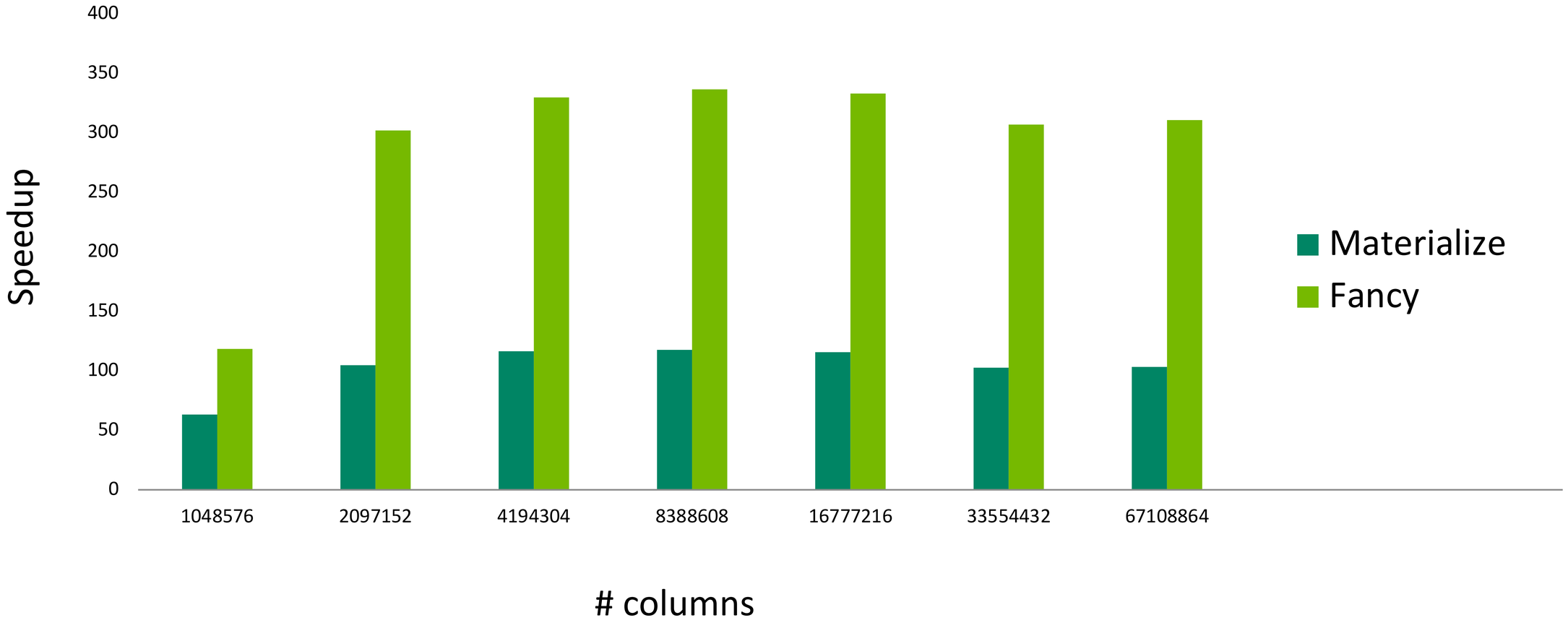
Solution

```
auto row_ids_begin = thrust::make_transform_iterator(
                     thrust::make_counting_iterator(0),
                     [=]__host__ __device__(int i) {
                       return i / width;
                     });
auto row_ids_end = row_ids_begin + temp.size();

thrust::universal_vector<float> sums(height);


thrust::reduce_by_key(thrust::device,
                      row_ids_begin, row_ids_end,
                      temp.begin(), thrust::make_discard_iterator(),
                      sums.begin());
```

N

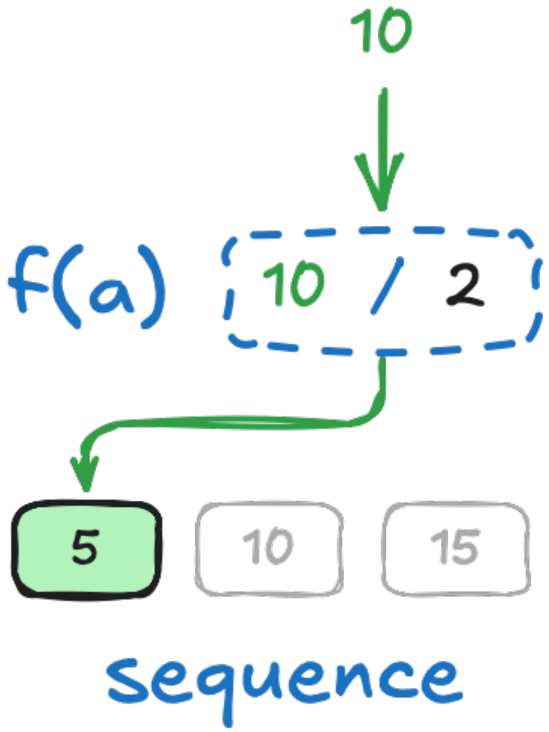
N

Serial vs Parallel



- Reduce by key is **100x** faster
- Fancy iterators + reduce by keys is **300x** faster

Transform Output Iterator




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

- Concept of iterators is not limited to input
- With another level of indirection, we can transform values that are written into transform output iterator

```
struct wrapper {
 int *ptr;

 void operator=(int value) { *ptr = value / 2; }
};

struct transform_output_iterator {
  int *a;

  wrapper operator[](int i) { return {a + i}; }
};


std::array<int, 3> a{ 0, 1, 2 };
transform_output_iterator it{a.data()};

it[0] = 10;
it[1] = 20;

std::printf("a[0]: %d\n", a[0]); // prints 5
std::printf("a[1]: %d\n", a[1]); // prints 10
```

Exercise: Segmented Mean

10 minutes

- Use transform output iterator to compute row mean

```
struct mean_functor {
  __host__ __device__ float operator()(float x) const {
    return x / width;
  }
};

thrust::universal_vector<float> means(height);
auto means_output = ...



thrust::reduce_by_key(thrust::device, row_ids_begin, row_ids_end,
                      temp.begin(), thrust::make_discard_iterator(),
                      means_output);
```


2N | 2N

01.05-Serial-vs-Parallel/01.05.03-Exercise-Segmented-Mean.ipynb

Exercise: Segmented Mean

Solution

```
struct mean_functor {
  __host__ __device__ float operator()(float x) const {
    return x / width;
  }
};

thrust::universal_vector<float> means(height);
auto means_output =
  thrust::make_transform_output_iterator(means.begin(), mean_functor{});


thrust::reduce_by_key(thrust::device, row_ids_begin, row_ids_end,
                      temp.begin(), thrust::make_discard_iterator(),
                      means_output);
```


N N

Memory Spaces

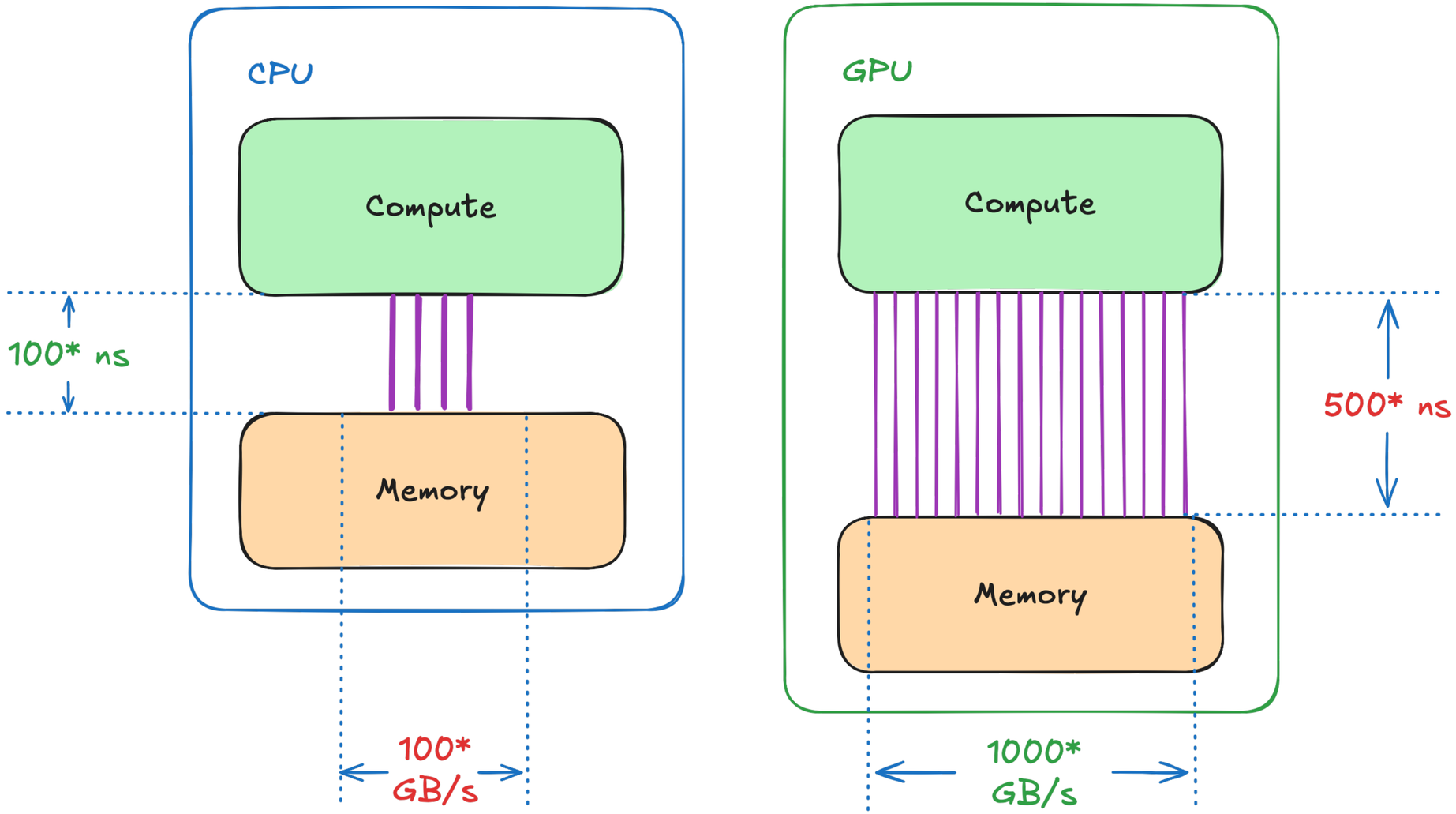
Problem statement

```
thrust::universal_vector<float> prev = dli::init(height, width);
thrust::universal_vector<float> next(height * width);

for (int write_step = 0; write_step < 3; write_step++)
{
  dli::store(write_step, height, width, prev);

  for (int compute_step = 0; compute_step < 3; compute_step++) {
    dli::simulate(height, width, prev, next);
    prev.swap(next);
  }
}
```

```
...
simulate 0.00016 s
  store
simulate 0.03123 s
simulate 0.00018 s
simulate 0.00018 s
  store
simulate 0.02954 s
simulate 0.00016 s
simulate 0.00016 s
...
```

- Why does computation step take 100x longer after storing?

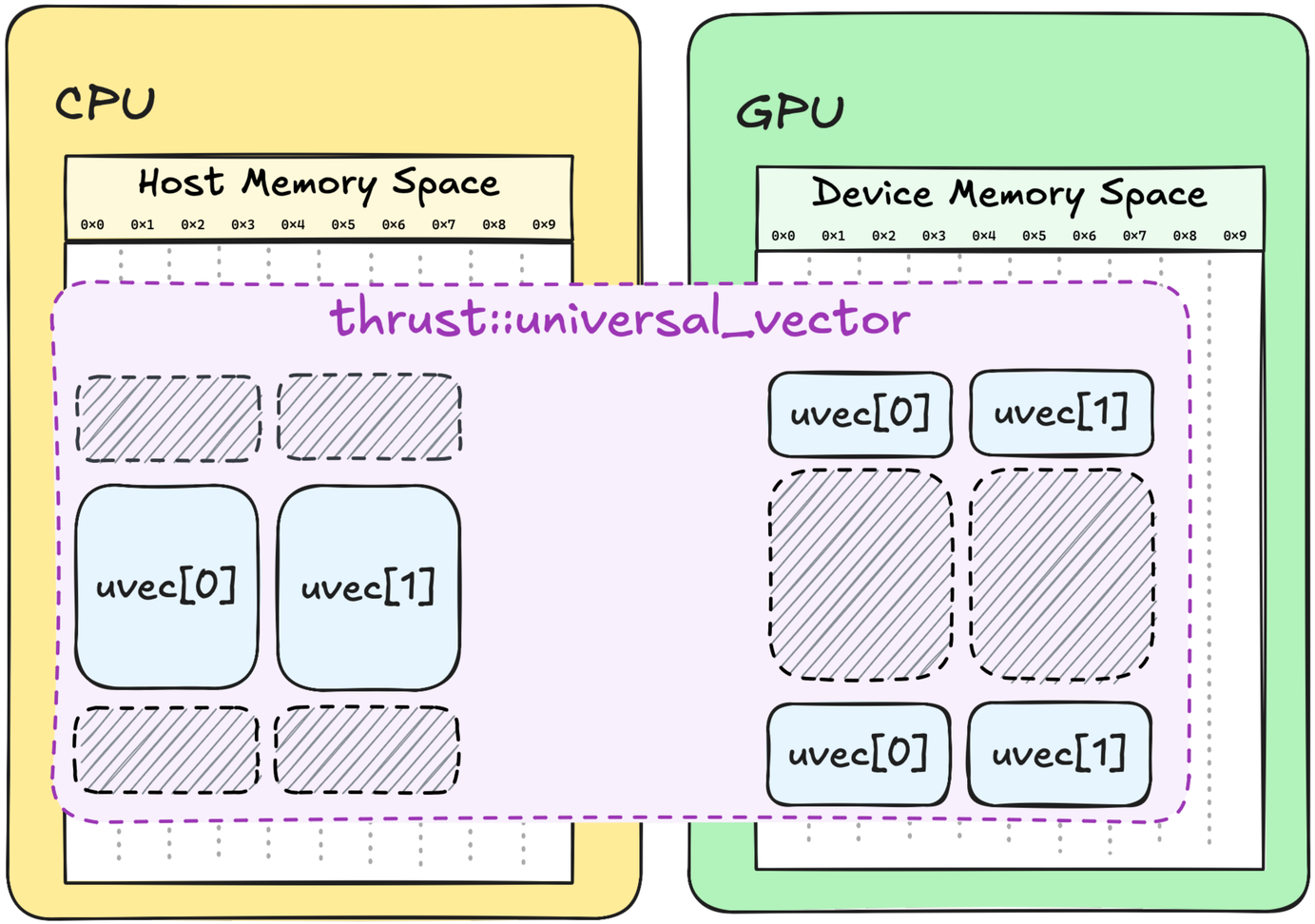
Memory Spaces



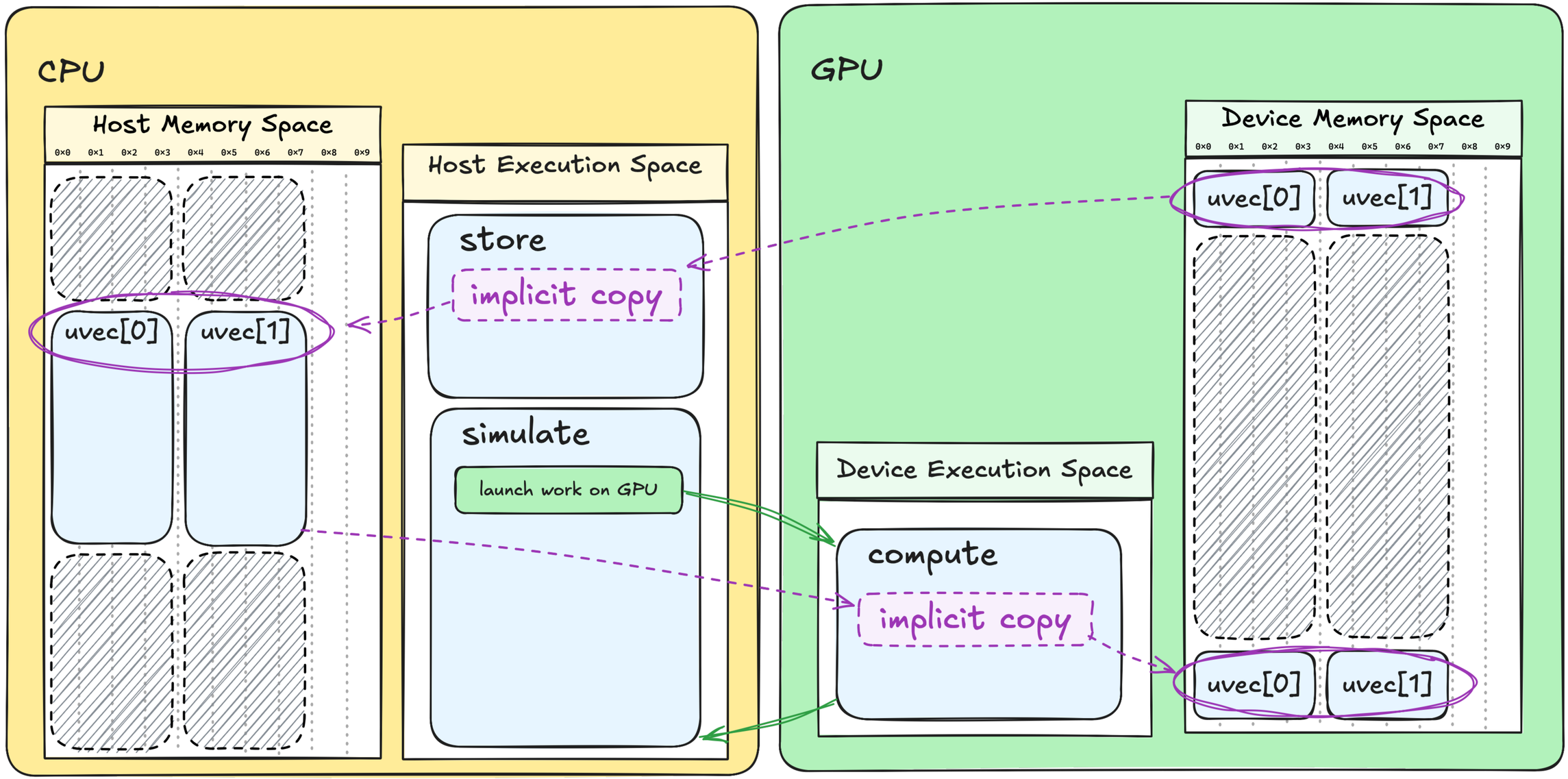
GPU has dedicated memory optimized for higher bandwidth

Memory Spaces

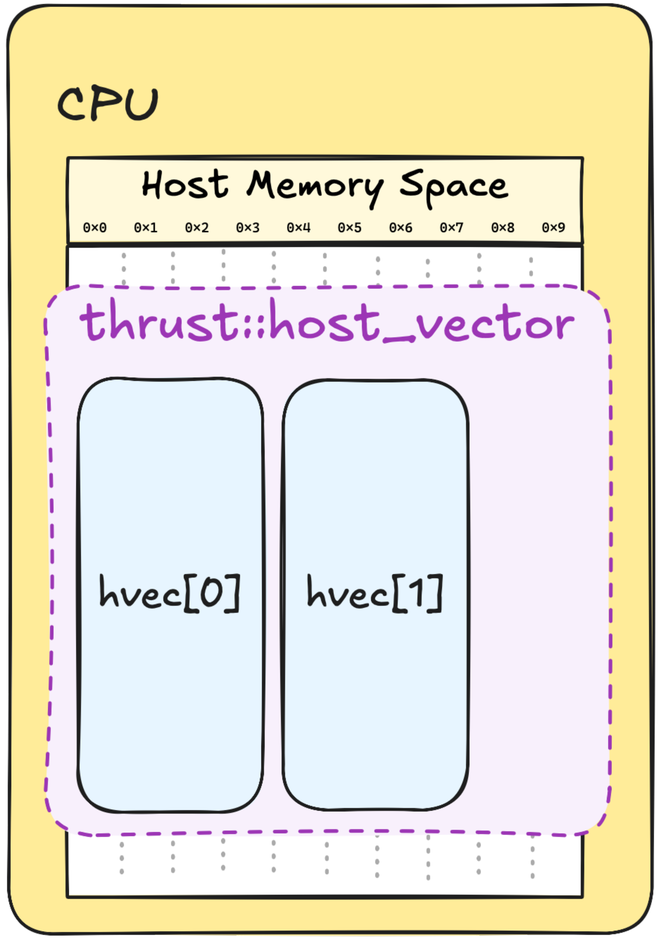
- Universal vector relies on managed memory
- Managed memory abstracts away the fact that GPU and CPU have distinct memory spaces
- Implicit memory transfer between memory spaces happens upon accessing memory



Memory Spaces



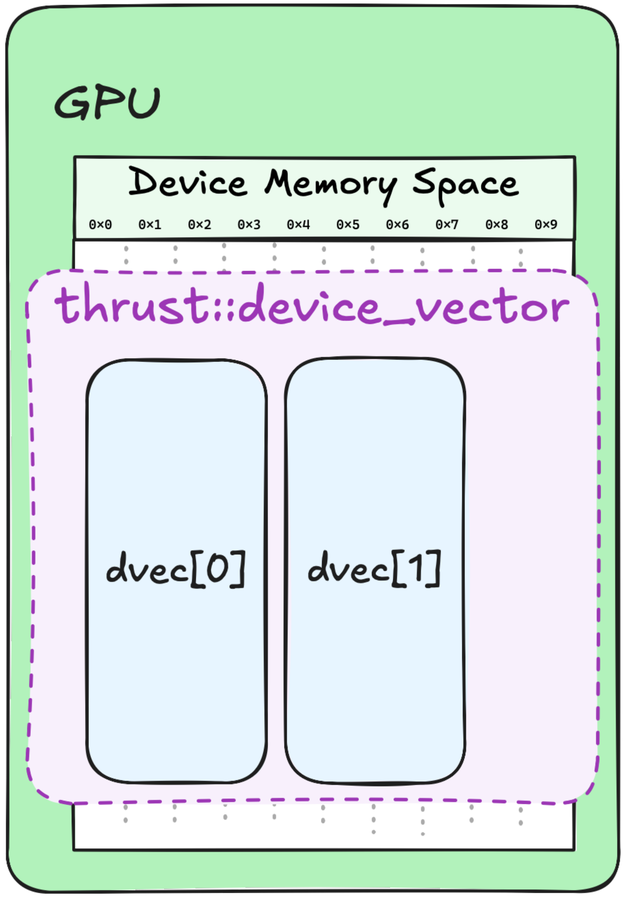
Memory Spaces



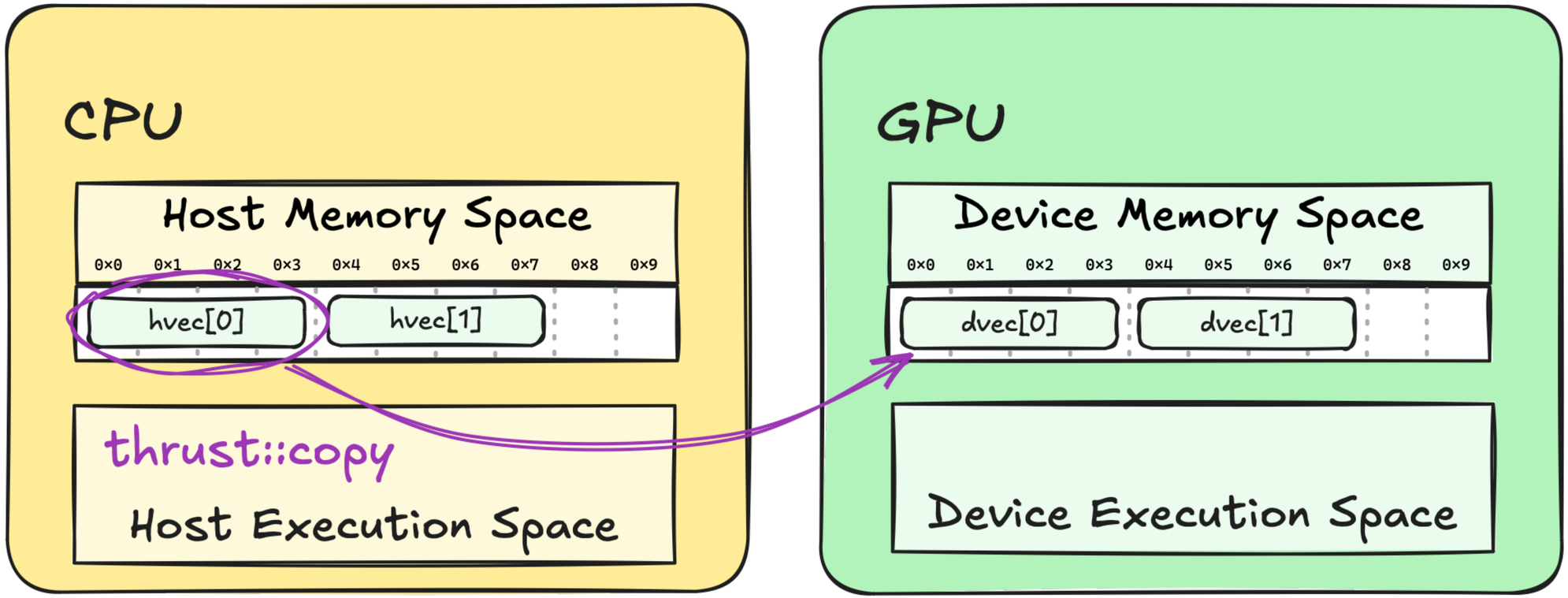
- Different containers allow you to be explicit in where you want memory

- **Host** vector:
  - constructed on **host**
  - allocates memory in **host** space
  - device cannot access it

- **Device** vector:
  - constructed on **host**
  - allocates memory in **device** space
  - host cannot access it



Memory Spaces



You can use copy algorithm to explicitly copy between memory spaces

Exercise: Use Explicit Memory Spaces

12 minutes

Use thrust::host_vector and thrust::device_vector instead of thrust::universal_vector

```
thrust::universal_vector<float> prev = dli::init(height, width);
thrust::universal_vector<float> next(height * width);


for (int write_step = 0; write_step < 3; write_step++)
{
  dli::store(write_step, height, width, prev);

  for (int compute_step = 0; compute_step < 3; compute_step++) {
    dli::simulate(height, width, prev, next);
    prev.swap(next);
  }
}
```

```
...
simulate 0.00016 s
  store
simulate 0.03123 s
simulate 0.00018 s
simulate 0.00018 s
  store
simulate 0.02954 s
simulate 0.00016 s
simulate 0.00016 s
...
```

01.06-Memory-Spaces/01.06.02-Exercise-Copy.ipynb

Exercise: Use Explicit Memory Spaces

Solution

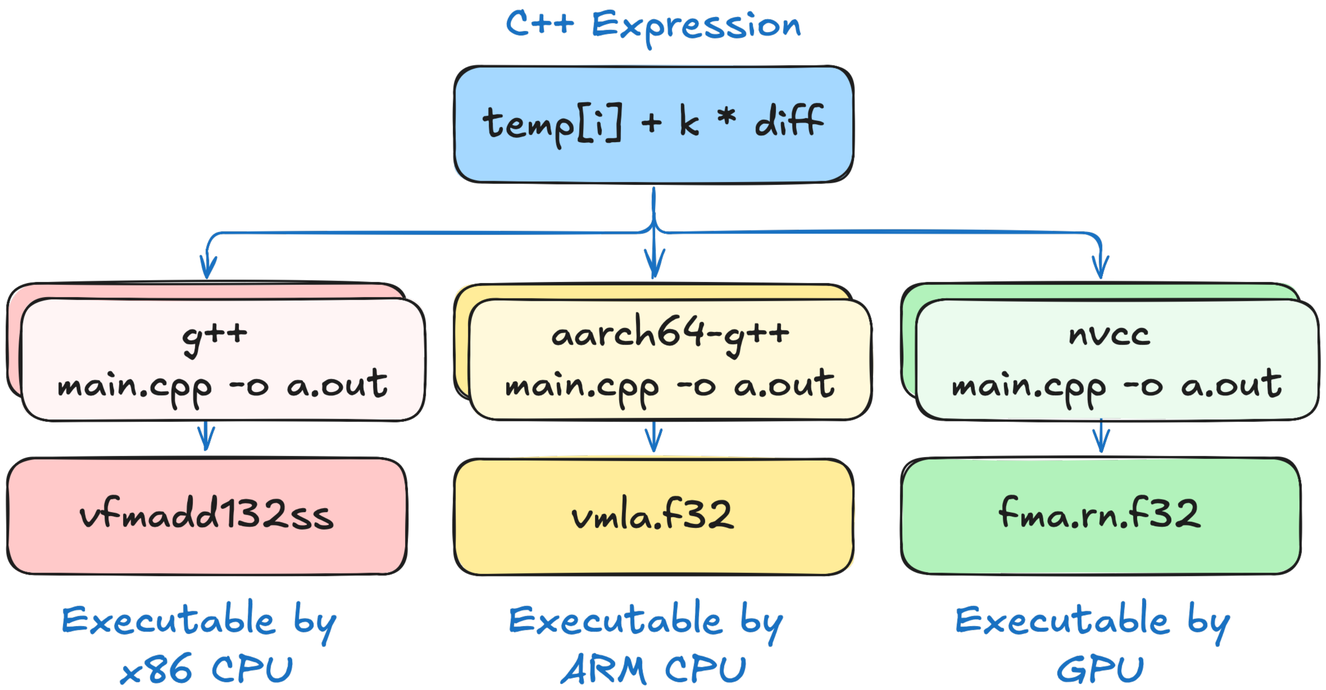
- All compute steps now take the same time to run

```
thrust::device_vector<float> d_prev = dli::init(height, width);
thrust::device_vector<float> d_next(height * width);
thrust::host_vector<float> h_prev(height * width);

for (int write_step = 0; write_step < 3; write_step++)
{
  thrust::copy(d_prev.begin(), d_prev.end(), h_prev.begin());
  dli::store(write_step, height, width, h_prev);

  for (int compute_step = 0; compute_step < 3; compute_step++)
  {
    dli::simulate(height, width, d_prev, d_next);
    d_prev.swap(d_next);
  }
}
```

```
...
simulate 0.00019 s
  store
simulate 0.00018 s
simulate 0.00017 s
simulate 0.00017 s
  store
simulate 0.00024 s
simulate 0.00016 s
simulate 0.00016 s
...
```

Takeaways

Use NVCC compiler

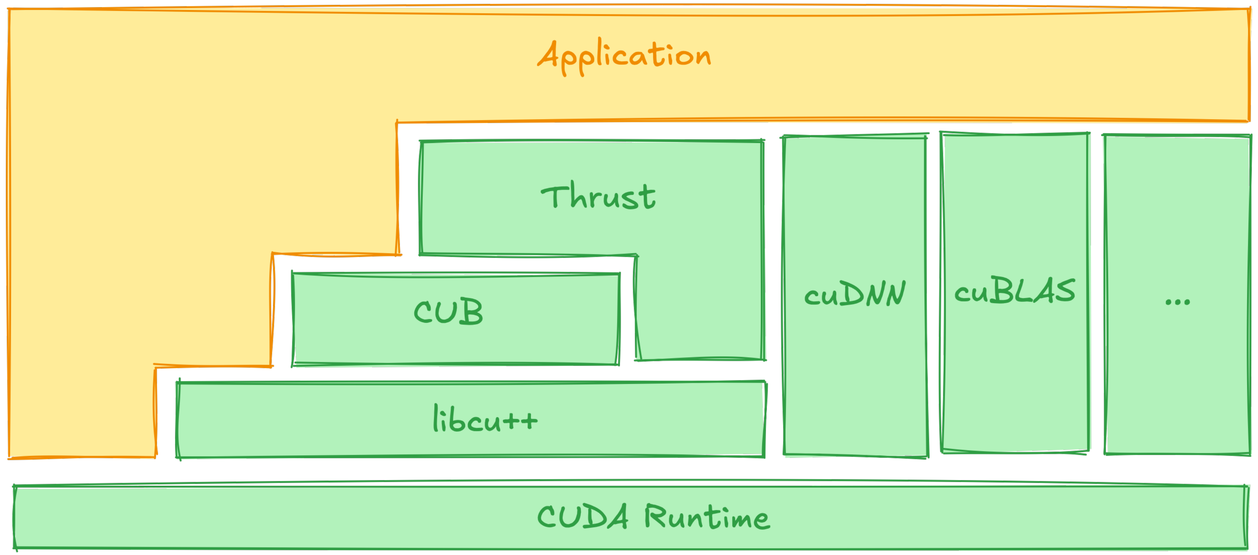


Takeaways

Use NVCC compiler

Use accelerated libraries as much as possible:

- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.



Takeaways

Use NVCC compiler

Use accelerated libraries as much as possible:

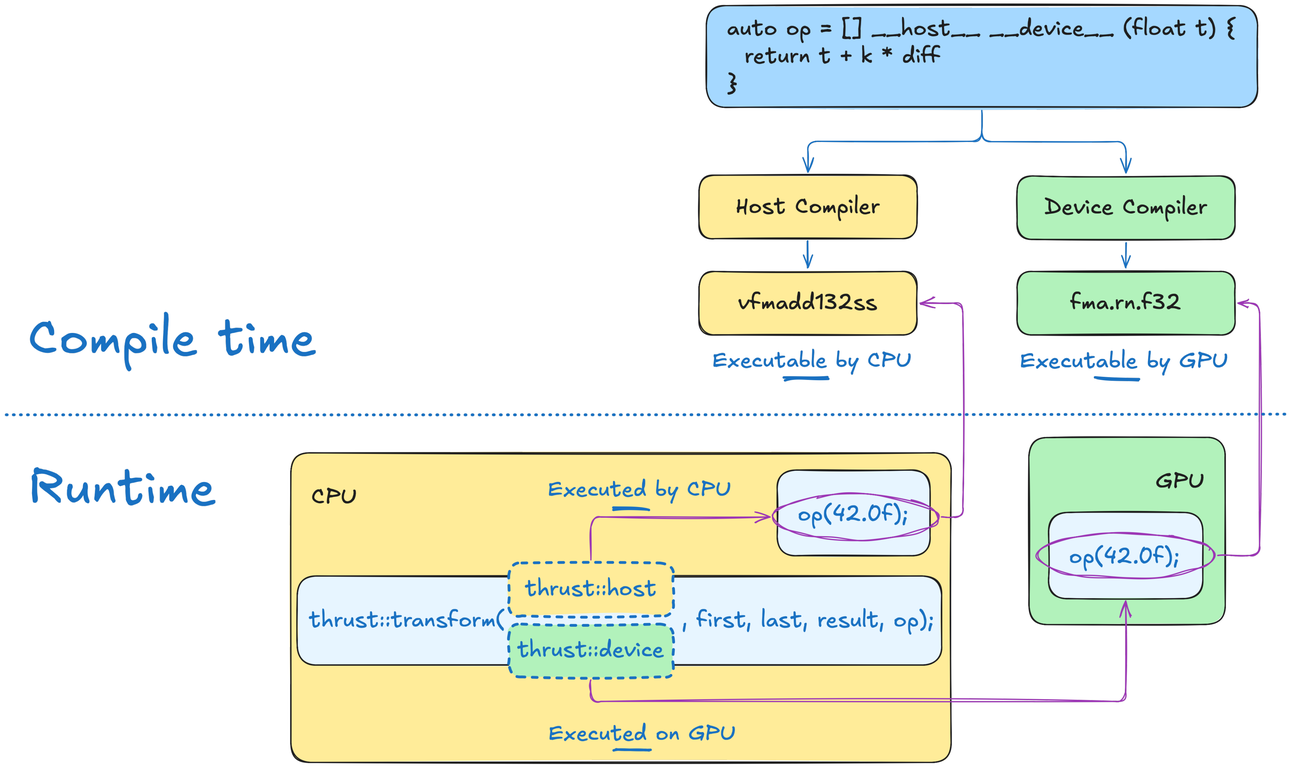
- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.

Use __host__ __device__ execution space specifiers:

- to specify where given function **could** be executed

Use thrust::host and thrust::device execution policies:

- to specify where given thrust algorithm **will** be executed



Takeaways

Use NVCC compiler

Use accelerated libraries as much as possible:

- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.

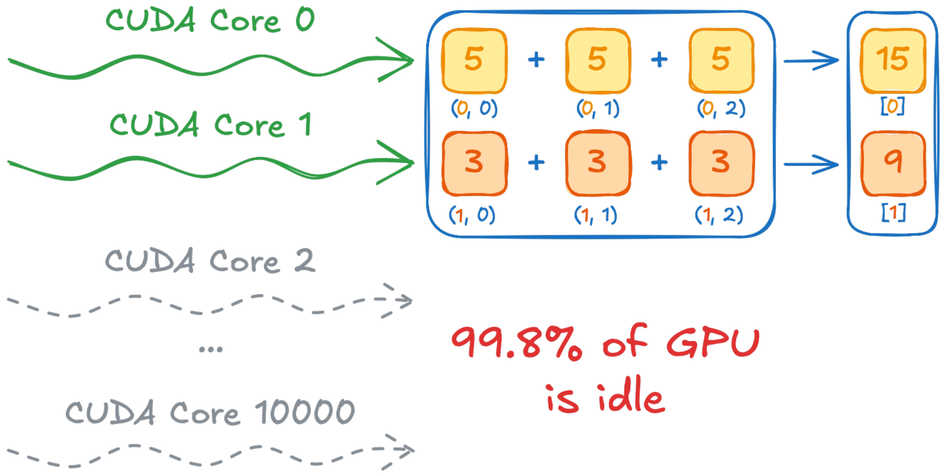
Use __host__ __device__ execution space specifiers:

- to specify where given function **could** be executed

Use thrust::host and thrust::device execution policies:

- to specify where given thrust algorithm **will** be executed

Avoid serialization as much as possible



Takeaways

Use NVCC compiler

Use accelerated libraries as much as possible:

- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.

Use __host__ __device__ execution space specifiers:

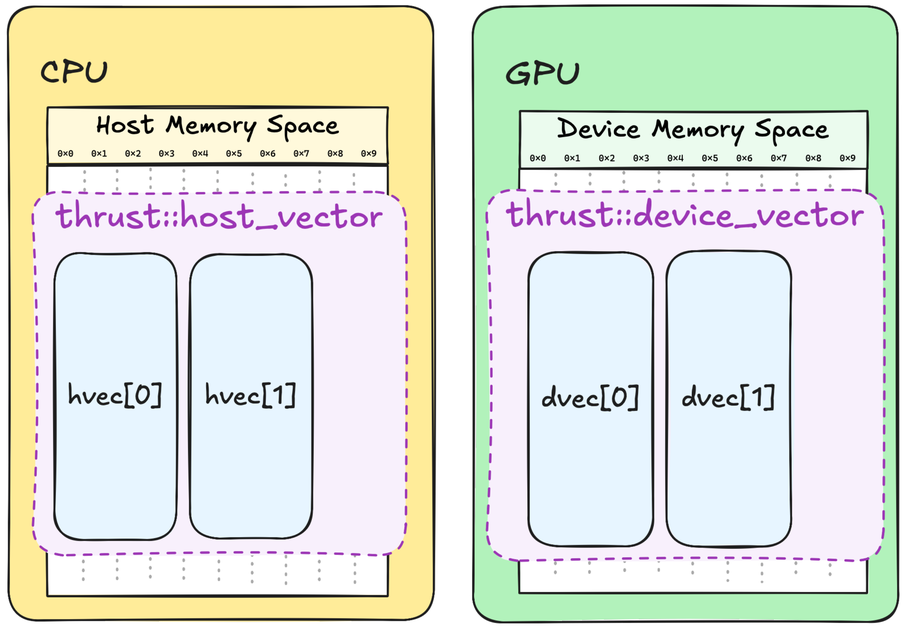
- to specify where given function **could** be executed

Use thrust::host and thrust::device execution policies:

- to specify where given thrust algorithm **will** be executed

Avoid serialization as much as possible

Use explicit memory spaces



Takeaways

Use NVCC compiler

Use accelerated libraries as much as possible:

- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.

Use __host__ __device__ execution space specifiers:

- to specify where given function **could** be executed

Use thrust::host and thrust::device execution policies:

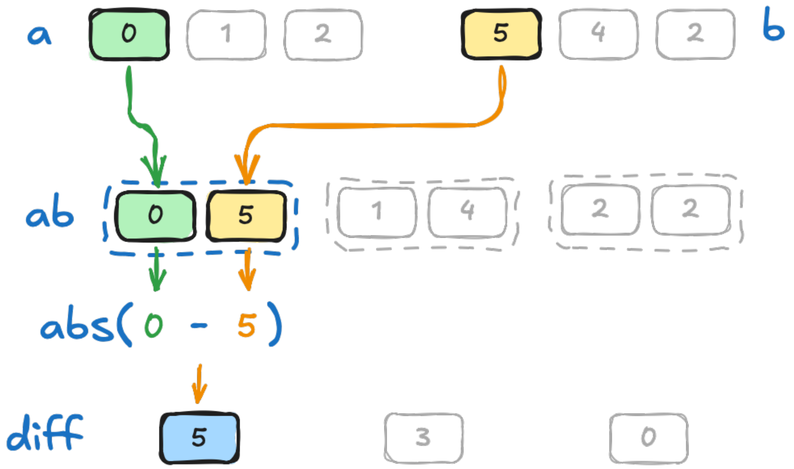
- to specify where given thrust algorithm **will** be executed

Avoid serialization as much as possible

Use explicit memory spaces

When existing libraries do not cover your use cases

- Use fancy iterators to extend them



Break

35 minutes