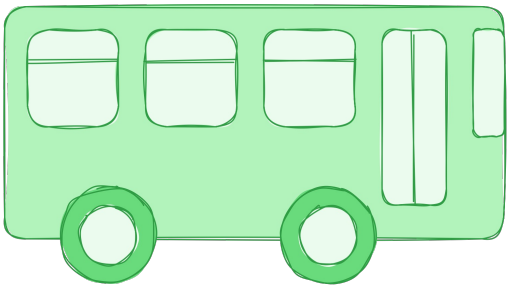
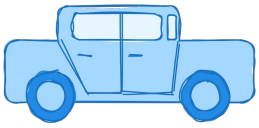
Welcome, everyone, and thanks for being here! If you've never programmed a GPU before, don't worry - you're in the right place. This session is designed for beginners, so no prior experience with CUDA or GPU programming is necessary. Before we dive in, let's talk about what we'll cover today. CUDA is a huge topic, and many of the concepts we'll touch on could fill entire courses. Because of that, we won't be able to explore everything in detail. Instead, our goal is to help you develop a solid intuition for how CUDA works. Think of this session as your introduction to the most important ideas and a chance to see the bigger picture of GPU programming.

GPU Architecture at a Glance



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

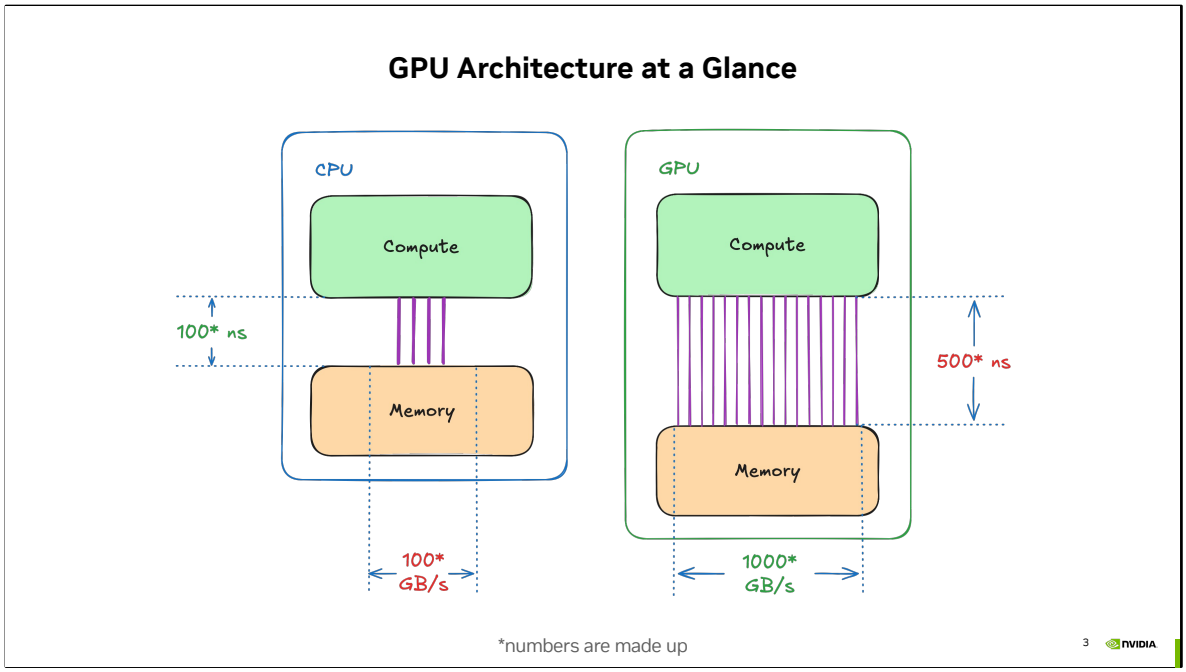
- Bus will take longer to move a few people
- Bus is optimized for moving many people



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

2 NVIDIA

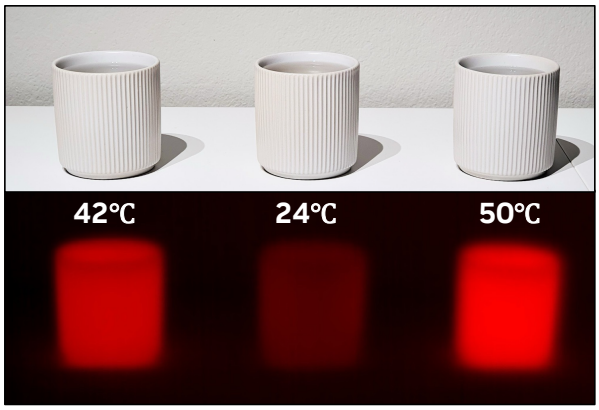
And we'll start by understanding why GPUs are so powerful. Picture a bus and a car. Which one is faster? This question is not precise enough. Faster in doing what? If you only have a few people to transport, a car is quicker and more efficient. But if you need to move a big crowd, a car will have to make multiple trips, taking more time overall. A bus, on the other hand, might be slower for a handful of passengers, but it's faster in carrying a large group all at once. That's exactly how CPUs and GPUs differ. CPUs are very fast and flexible when handling a small number of tasks. GPUs, on the other hand, can handle huge numbers of tasks in parallel, making them perfect for large datasets. CUDA is the tool that lets us leverage GPUs for the right kinds of problems.



Next, let's see how this bus-and-car analogy applies to real world. On this slide, you can see two key parameters: bandwidth and latency. **Bandwidth** is how much data you can move at once. **Latency**, on the other hand, is how long it takes to move a single piece of data. In our simplified example, the CPU (on the left) has lower latency: it can quickly access a single byte (say, in 100 ns). The GPU (on the right) takes longer (about 500 ns), so it's slower in loading a single byte. But the GPU makes up for it with much higher bandwidth - up to 10 times more in this case. It's like a bus that may not be as fast as a car for one or two passenger but can carry far more people in a single trip. Real hardware will have different numbers, but the key idea remains. GPUs excel when you need to process lots of data in parallel.

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



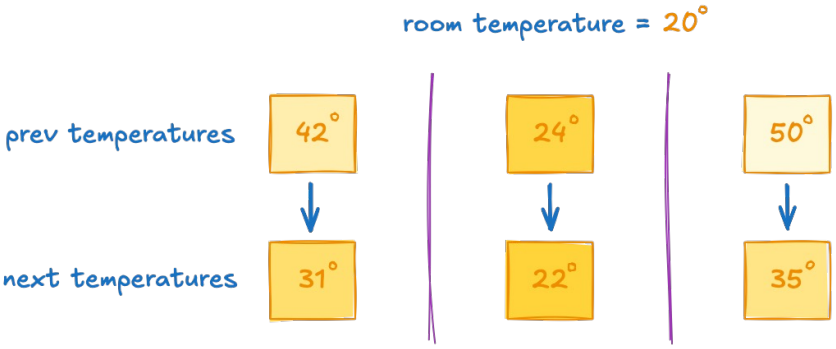
4 NVIDIA

So, how do you know if your problem will benefit from GPU's higher bandwidth? To build some intuition, imagine you have three cups of water at different temperatures: 42°, 24°, and 50°. You want to simulate how they cool down over time. But simulating every tiny detail - like shape or material - would be too complex. Instead, you can simply represent each cup with just one number: its temperature. Think of it like placing a temperature sensor in each cup.

NVIDIA

Which Problems Benefit from Bandwidth?

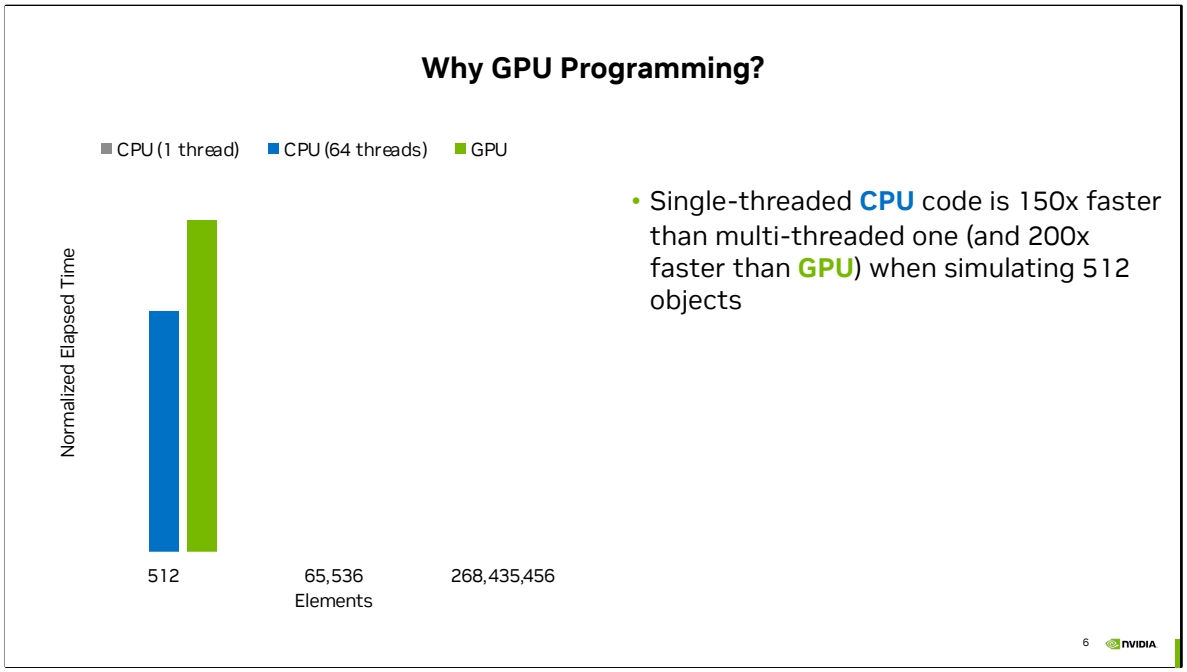
- Let's say we want to simulate **N** objects cooling down to a room temperature
- Can this problem benefit from immense bandwidth provided by GPUs?
- Just like before, the question is not precise enough
- How large is **N**?



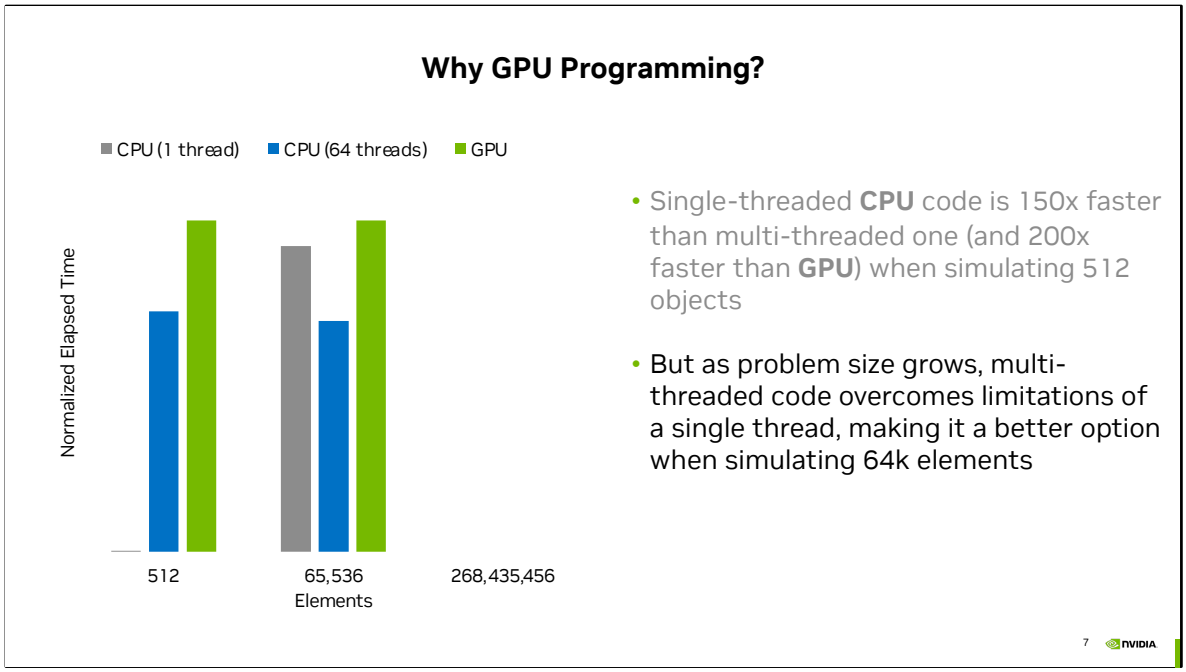
5 NVIDIA

Now the problem is much simpler: we have objects, each with its temperature. And we want to simulate how these objects cool to the room temperature. The question is: will this problem benefit from a GPU? Just like with the bus-and-car analogy, this question is not precise enough. It depends on how many objects you're dealing with. In other words, the answer depends on `N`.

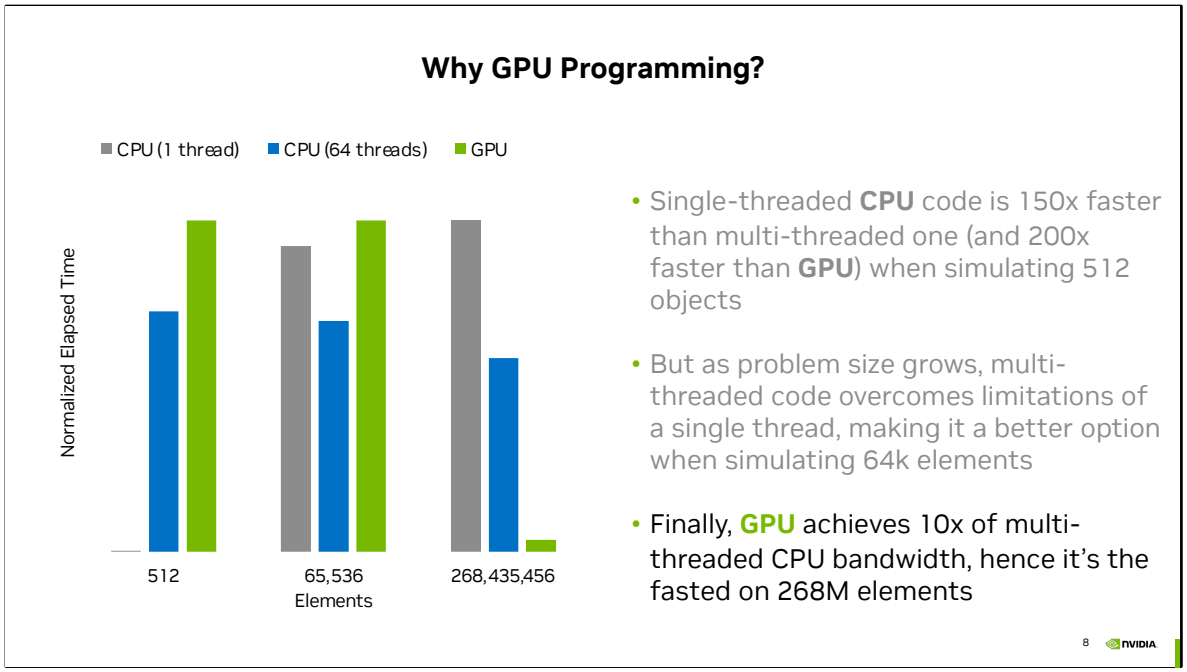
NVIDIA



Let's start small and see what happens when N is equal to 512. While this may be a lot of cups for a kitchen, it's actually too little data to take advantage of parallelism on either the GPU or the CPU to that matter. On this slide you can see the time it take to solve the problem. So, the lower the bar the better. In this scenario, single-threaded CPU code runs so quickly that we can't even see the bar representing it. In fact, here single-threaded CPU code is 150 times faster than multi-threaded code and 200 times faster than the GPU. This happens because the overhead of launching threads and managing parallelism outweighs the benefits of increased throughput. For such small workloads, a single-threaded CPU approach is usually the most efficient.

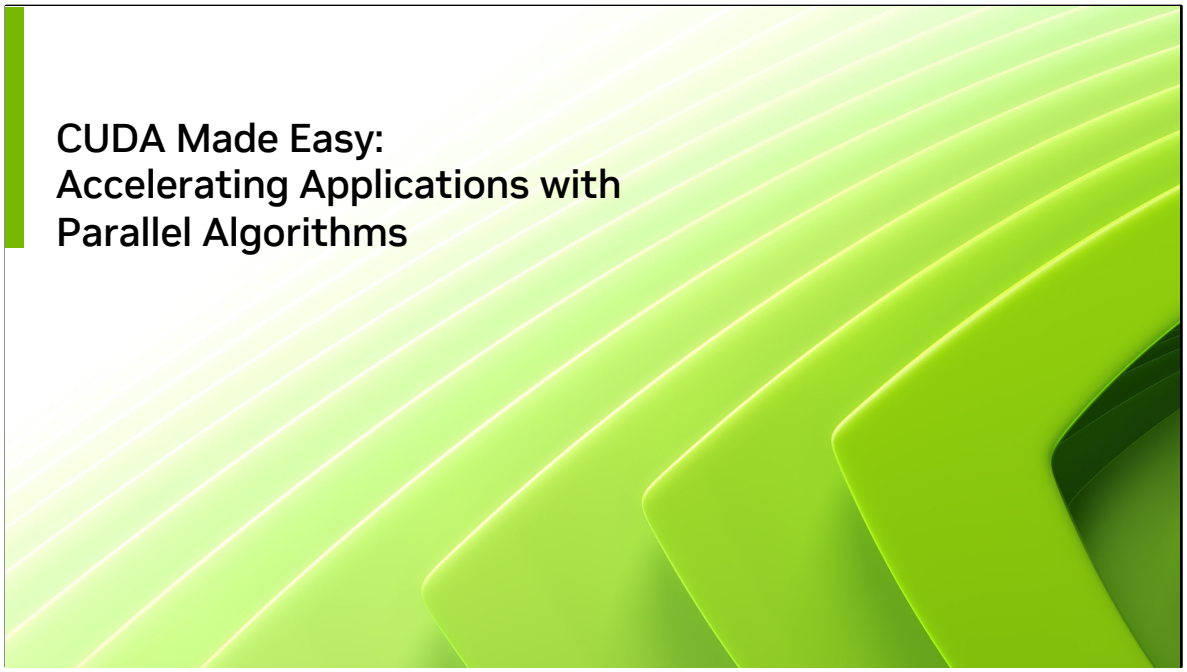


Now let's scale up to 65,000 objects. At this size, the multi-threaded CPU code finally overtakes the single-threaded approach. Increased memory and arithmetic throughput of multi-threaded CPU code finally overcomes mentioned overheads. But we can't say this about GPU yet. At this problem size, it performs similar to the single-threaded code.

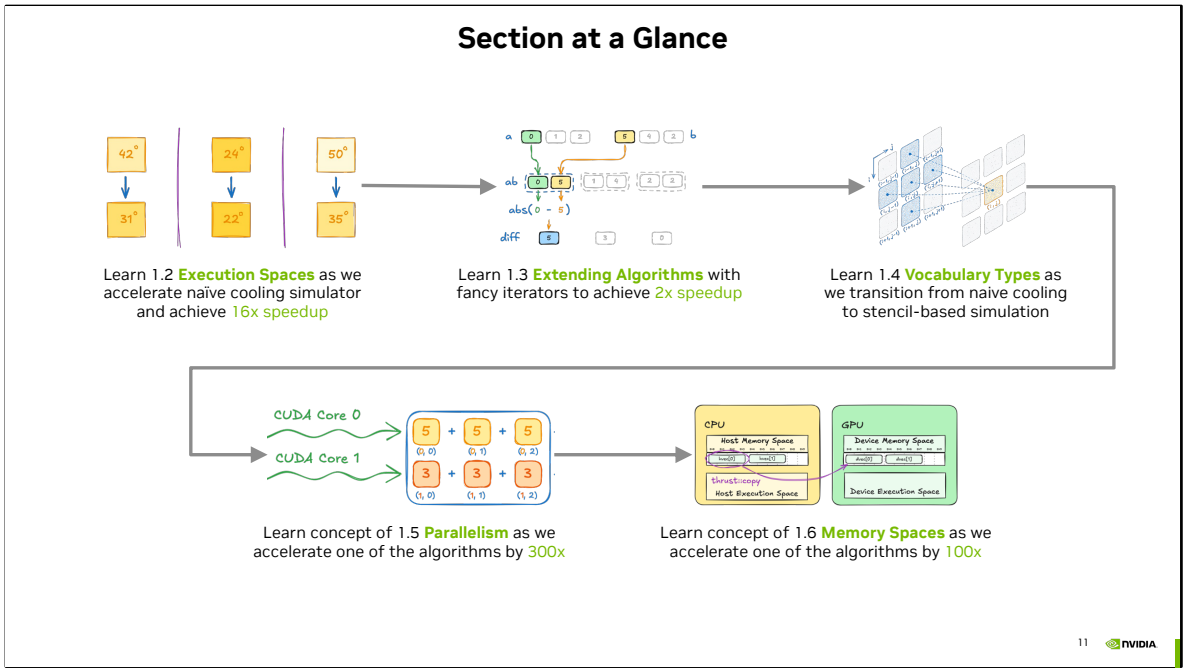


Finally, if we jump to 200 million objects, the GPU takes the lead, performing roughly ten times faster than the multi-threaded CPU code. That ten-fold improvement directly reflects the difference in bandwidth between CPU and GPU. But, as you can see, you need a large enough problem to take advantage of this increased bandwidth.

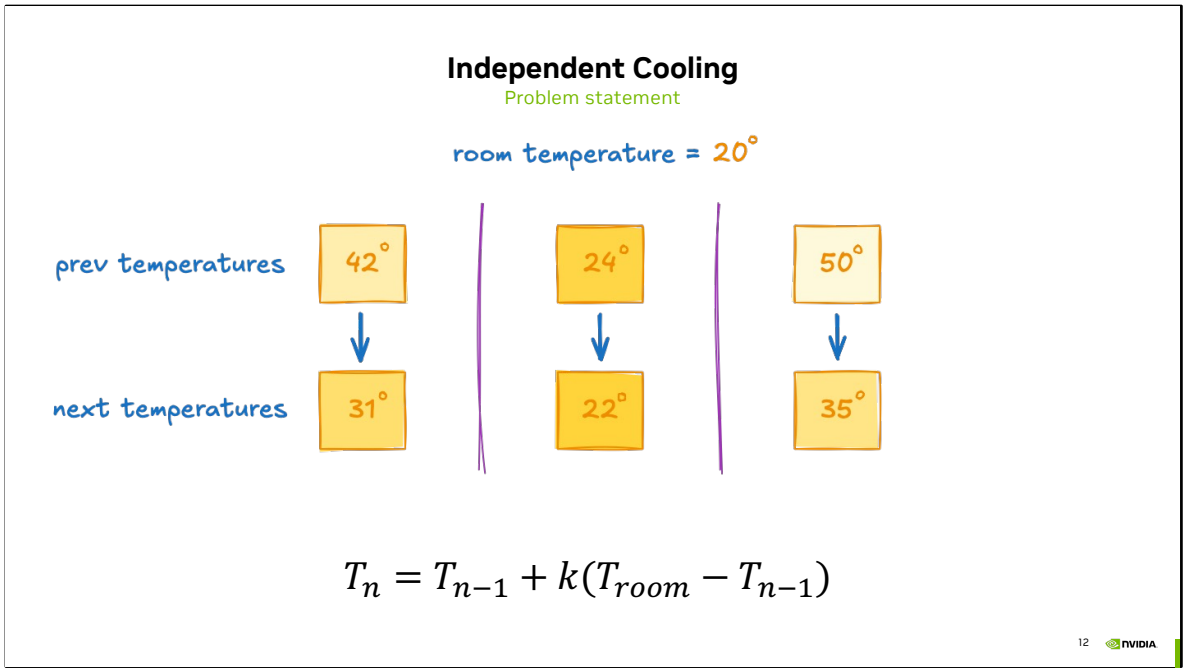
With that in mind, let's take a look at our agenda for today. First, we'll begin with "CUDA Made Easy" section. Here, you'll learn the basics of writing CUDA code. You'll get hands-on practice by porting the cooling simulator we mentioned from the CPU to the GPU. After that, we'll move on to the next section, called "Unlocking the GPU's Full Potential". Here, you'll learn a few optimization techniques allowing you to take full advantage of the GPU's. Then, we'll dive into "Implementing New Algorithms with CUDA Kernels". In this section, you'll explore designing and building custom GPU algorithms for problems not covered by existing libraries. Finally, we'll wrap up with a final assessment that ties everything together, giving you a chance to apply what you've learned in a real-world context. By the end of the day, you'll have a solid foundation to start writing your own GPU-accelerated programs and to continue your journey into CUDA.



Let’s get started with the first section! Here, we’ll focus on the fundamentals of CUDA programming. Our goal is to demystify the process of writing programs that run on GPUs and show you how to think in terms of parallel algorithms.



Let's take a look at what we'll cover in this section. We'll begin with the essential concept of *execution spaces*, which will enable us to GPU-accelerate our simple cooling simulator using parallel algorithms. Next, we'll explore how to extend these parallel algorithms when they don't precisely fit your needs. Along the way, we'll introduce the concept of *vocabulary types* to boost developer productivity. We'll apply these vocabulary types to convert our basic cooling simulator into a full 2D heat equation simulator. As we do that, we'll build a clear understanding of how serialization impacts performance. Finally, we'll discuss *memory spaces* and see how they can help us further optimize the simulator. By the end of this section, you'll have a solid grasp of how to GPU-accelerate your code with parallel algorithms.



Let's revisit the problem of N objects cooling down to room temperature. Our goal is to simulate how these objects gradually approach the ambient temperature, using the equation shown below. At each step, the equation adjusts an object's temperature based on the difference between its current temperature and the room temperature. Crucially, each object cools down independently. They all follow the same rule, but one object's temperature doesn't affect another's. This independence makes it a perfect candidate for parallel computing, because we can update every object's temperature at the same time. Below, you can see how each object's temperature changes at the next step. Over time, they all move closer to the ambient temperature of 20 degrees.

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)

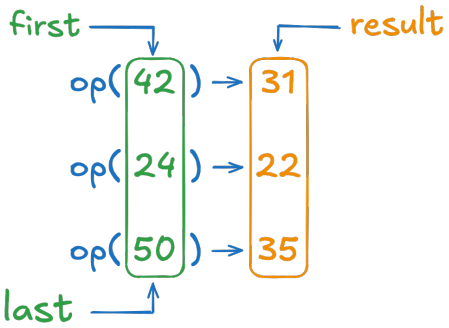
step	temp[0]	temp[1]	temp[2]
0	42.0	24.0	50.0
1	31.0	22.0	35.0
2	25.5	21.0	27.5

13 NVIDIA

Let's see how we can implement this simulation in C++. We start by defining a cooling coefficient, k, and setting ambient temperature of 20 degrees. The temp vector holds initial temperatures for three objects. We then define a lambda function that calculates the new temperatures based on the old ones. Next, we apply this function to each temperature using std::transform, which updates the values in-place. By running this transformation for several steps, each object's temperature gradually shifts closer to ambient. This can be seen in the table on the right. Initially, on the step zero, we have temperatures we started with, 42, 24, and 50 degrees. Towards the second step, these temperatures turn into 25, 21, and 27 degrees.

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)

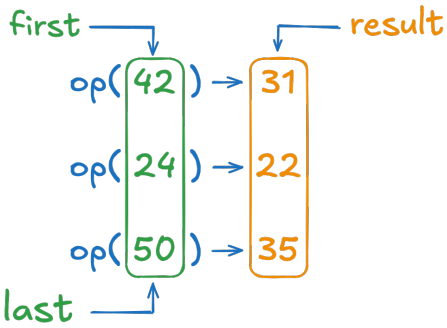
14 NVIDIA

Just in case you're not familiar with std::transform, here's a quick overview. It's a standard library algorithm that takes a range of input elements, applies an operation to each one, and stores the result in a given output range. Think of it as a conveyor belt: each item goes in, gets transformed, and then the output is placed somewhere. In our case: we provide the start and the end of the temp vector to specify the input range; we then specify temp.begin() again as the output, so we overwrite the original values in-place; finally, we pass our lambda function, which calculates the new temperature. Visually, each old temperature is fed into the operation: 42 becomes 31; 24 becomes 22; 50 becomes 35.

NVIDIA

Transform

For execution on GPU



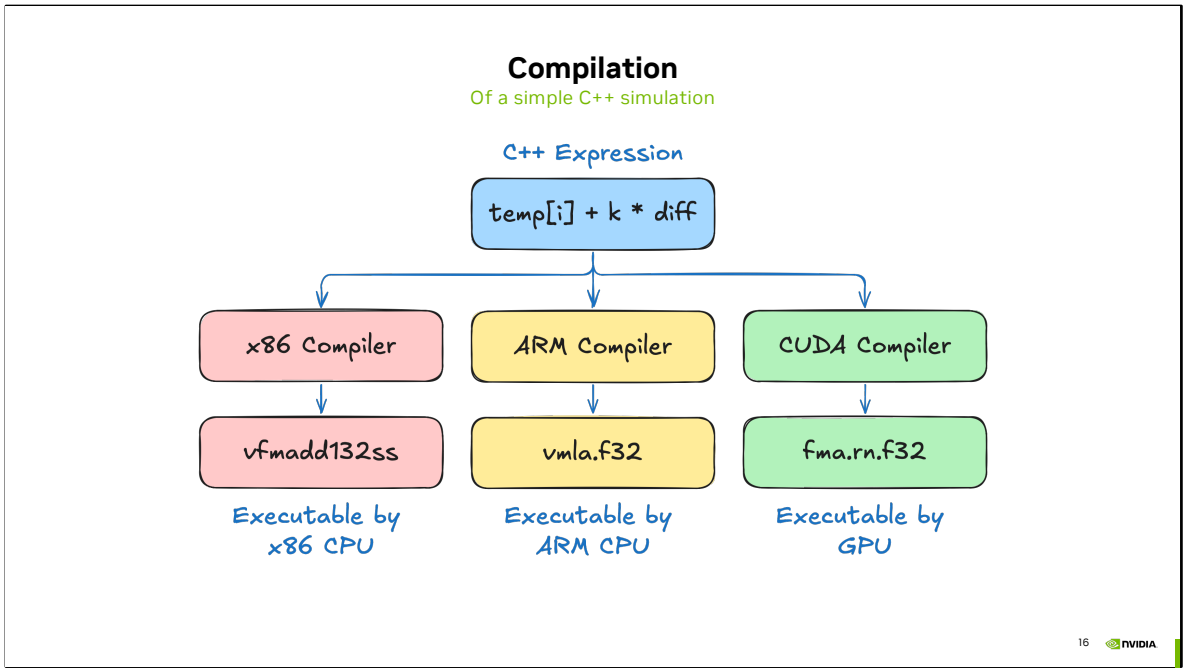

```
auto op = [=](float temp) {
  float diff = ambient_temp - temp;
  return temp + k * diff;
};

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)

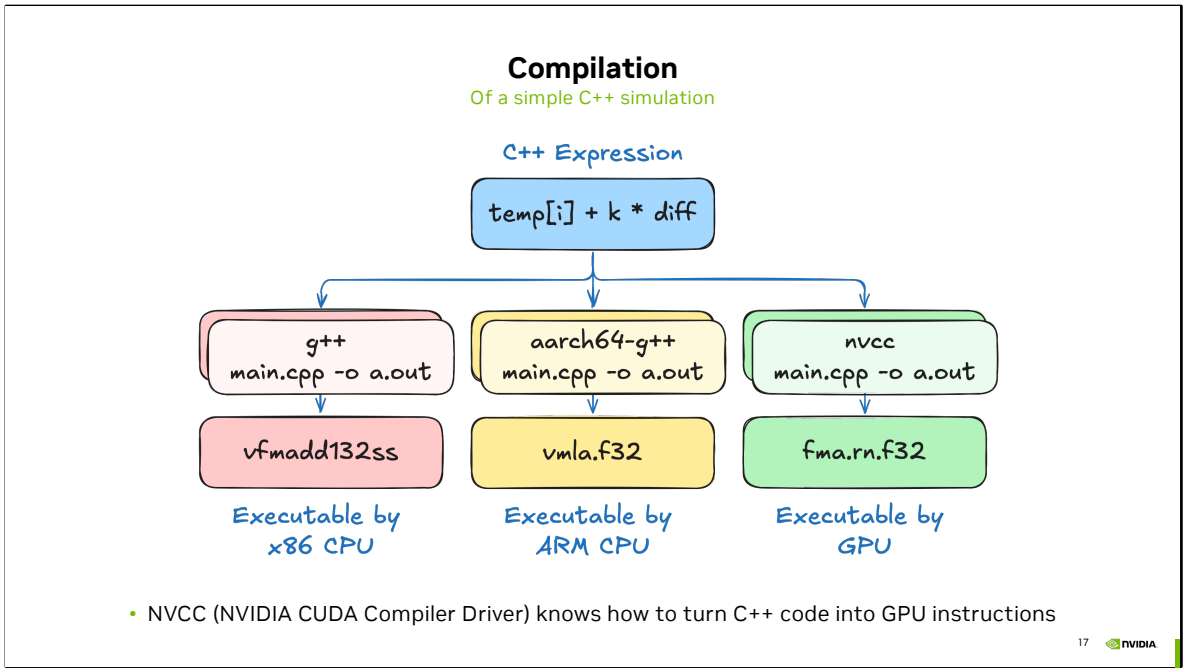
15 NVIDIA

In C++, std::transform can be implemented as a simple for loop, like the one you see on the screen. But let's take a step back and see what must happen to run this code on the CPU.

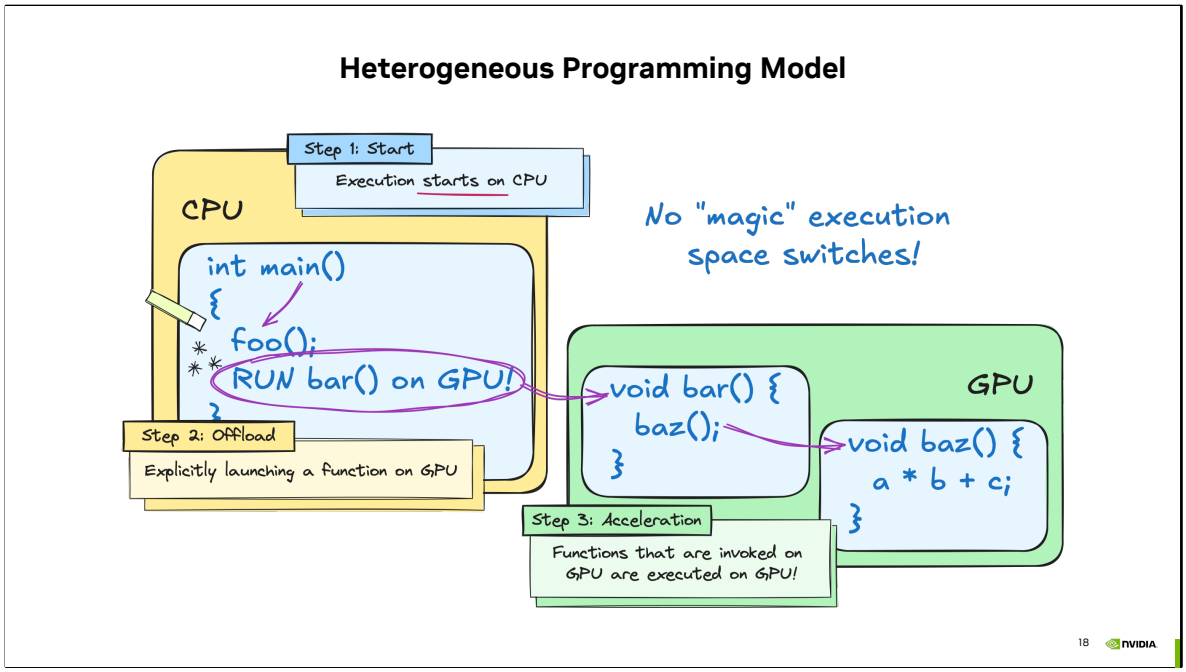
NVIDIA



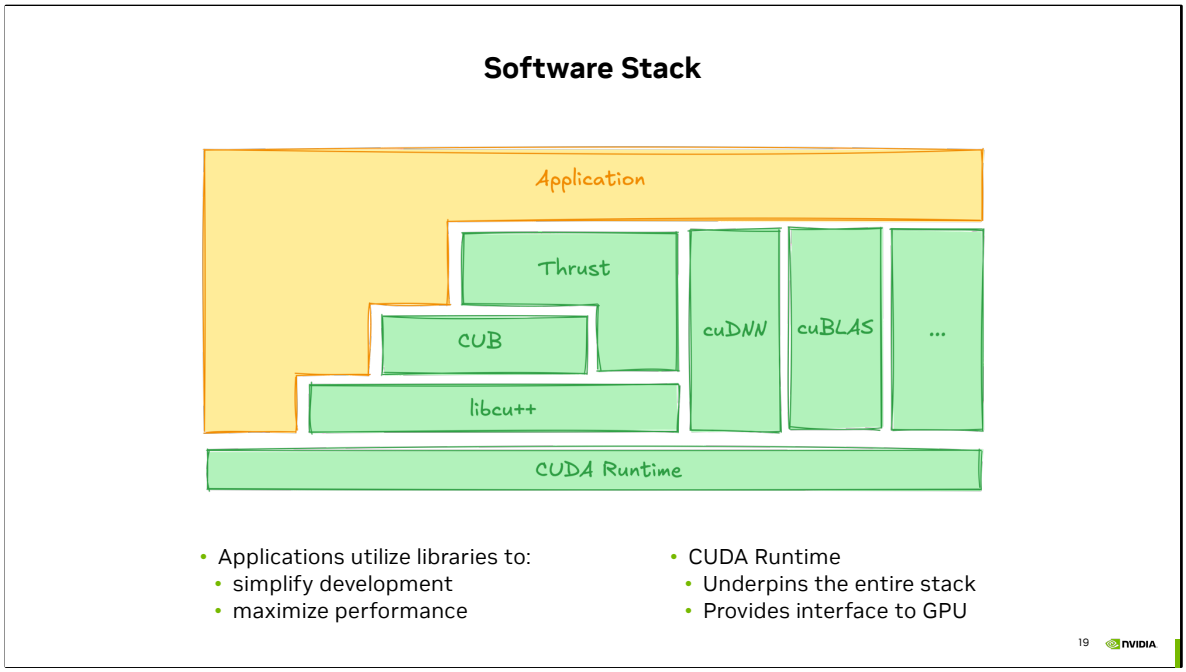
The problem is, CPUs do not run C++ code! They only understand machine instructions. That's why we need a compiler to translate our C++ code into CPU instructions. For instance, C++ expression on the slide will likely be compiled into fused multiply-add instruction like vfmadd. However, different CPUs have different instruction sets. For example, vfmadd instruction only works on x86 CPUs. Other CPU architectures, say ARM, have different instructions. This means that by default, the code you compile for one CPU architecture like x86, won't run on another architecture, like ARM, and vice versa. And GPUs are no different. They're just another processor type with their own set of instructions. This means we need a specialized compiler that can emit GPU instructions.



That's where the NVIDIA CUDA Compiler, or NVCC, comes in. NVCC is a tool that lets us compile CUDA C++ code specifically for NVIDIA GPUs. Congratulations, you've just compiled your first CUDA program! But there's one issue. The program you've just compiled is not using GPUs at all. That might be surprising because when we compiled out code for CPU, the entire program was executed on a CPU. But now we compile our program for GPU, and nothing runs on GPU. This confusion is an indicator that we are missing an important piece of CUDA programming model.



GPUs are accelerators rather than standalone processors. A lot of work, like interactions with network and file system, can only be done on CPU. So, a CUDA program always *starts* on the CPU. You are responsible for explicitly specifying which code has to run on the GPU. In other words, you are responsible for specifying which code runs **where**. The established terminology for **where** code is executed is **execution space**. At a high level, these execution spaces are partitioned into **host** (CPU) and **device** (GPU). These terms are used to generalize programming model. Something other than a CPU could host a GPU, and something else than a GPU could accelerate a CPU. By default, code runs on the **host** side. You are responsible for specifying which code should run on the **device**. This should explain why using NVCC alone was insufficient. We haven't marked any code for execution on the GPU.



So, how do we actually run any code on the GPU? In C++, we often rely on libraries rather than re-implementing every feature from scratch. For instance, it's likely that you've used printf or std::sort instead of implementing them. And CUDA is no different. Along with NVCC, CUDA provides a wealth of GPU-accelerated libraries. Some libraries are generic, like **Thrust**, which offers high-level algorithms similar to the C++ Standard Library. There are also domain-specific libraries, such as **cuDNN** for deep neural networks and **cuBLAS** for linear algebra. Most of these build on the CUDA Runtime - the core set of APIs for managing and communicating with the GPU. Since this course is not domain-specific, we'll focus on one of the generic libraries – Thrust. Let's dive in!

Thrust Overview

Standard Algorithms

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

Containers

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

Function Objects

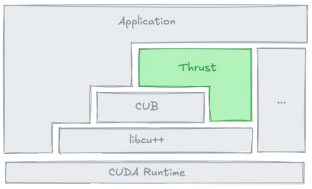
- `thrust::plus`
- `thrust::less`
- `thrust::multiplies`
- `...`

Extended Algorithms

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

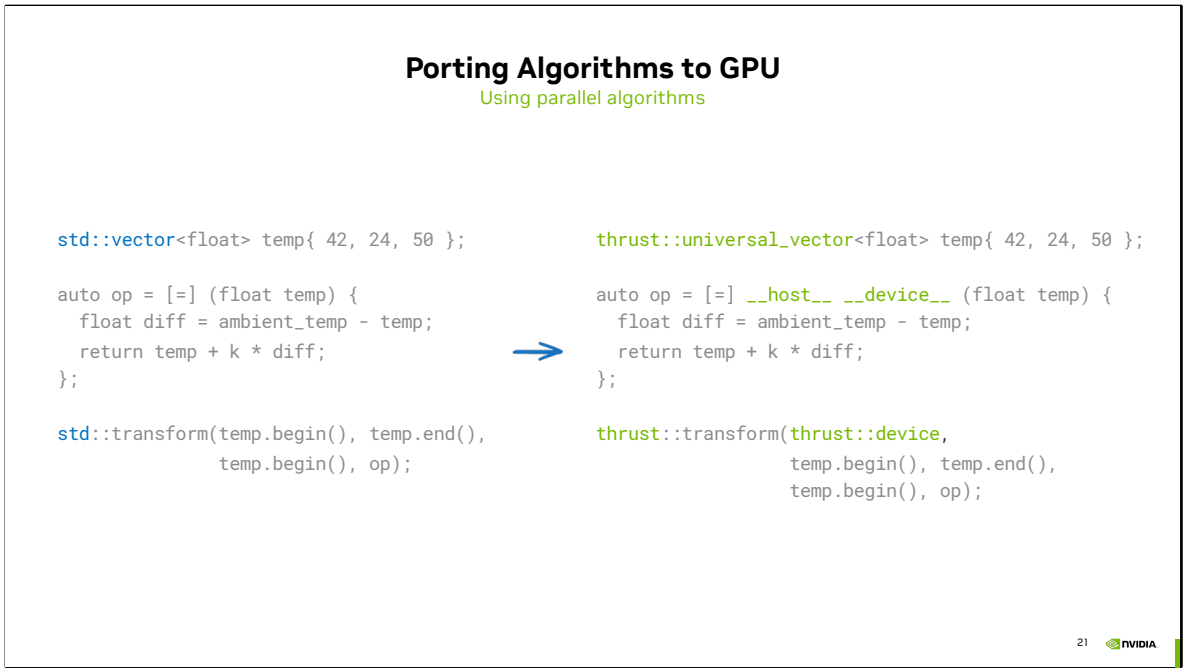
Iterators

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



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

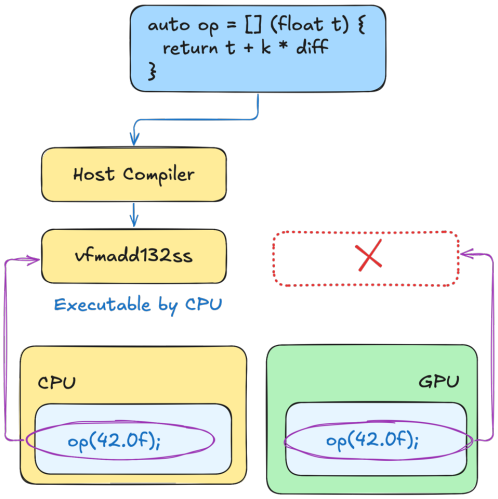
Thrust is a library that helps you write GPU-accelerated programs using tools that feel very similar to the C++ Standard Library. It simplifies working with GPUs by offering prebuilt components for common tasks: Thrust containers give you GPU-friendly alternatives to things like std::vector. And Thrust algorithms, like thrust::copy, thrust::sort, etc., provide GPU-accelerated alternatives to standard algorithms. Long story short, Thrust makes it easier to write clean, efficient code for GPUs without worrying too much about the low-level details. We'll start simple and explore how Thrust can help accelerate our cooling simulator.



On the left, you can see our CPU version, where we use std::vector to store data and std::transform to apply the cooling operation. On the right, we've adapted it for GPU execution with Thrust: we replaced std::vector with thrust::universal_vector; we also added __host__ __device__ specifiers to the lambda; and lastly, we swapped std::transform for thrust::transform, and provided the thrust::device execution. With these small changes, our code is now set up to run efficiently on the GPU. Thrust keeps the style of the C++ standard library but makes GPU acceleration much more straightforward. Next, let's take a closer look at each of these adjustments, starting with the execution space specifiers.

Porting Algorithms to GPU

Execution space specifier

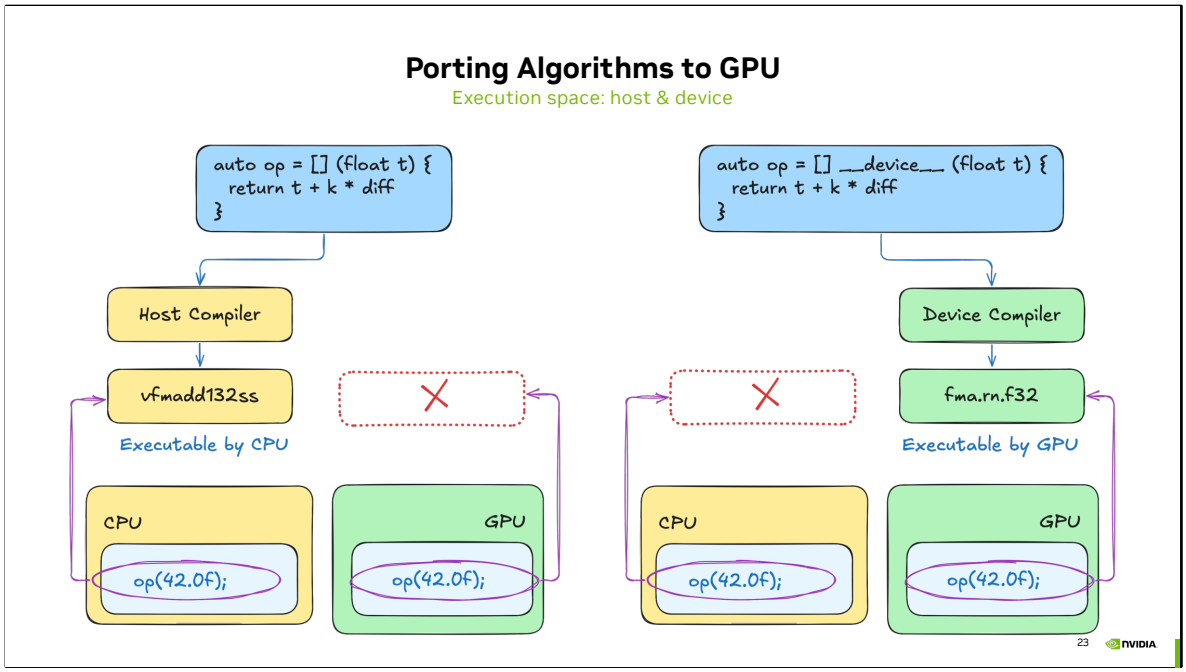


- By default, all functions are compiled for CPU
- If you attempt to invoke this function on GPU, there'll be no code for it to execute

22 NVIDIA

As we discussed, you must specify which hardware each function is compiled for. That's where *execution space specifiers* come into play. If you remember, all code is compiled for CPUs by default. That means that if you don't annotate the function in any way, there'll be no instructions for GPU to execute!

NVIDIA

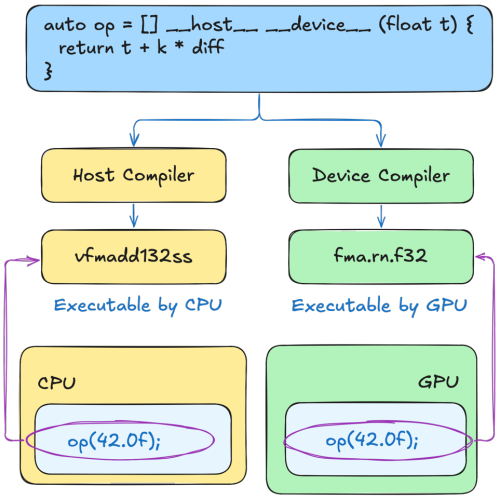


To fix that, you can annotate lambda function with `__device__` execution space specifier. In this case, nvcc will generate instructions only for the GPU. But now, there are no instructions for the CPU to execute.

NVIDIA

Porting Algorithms to GPU

Execution space: host-device

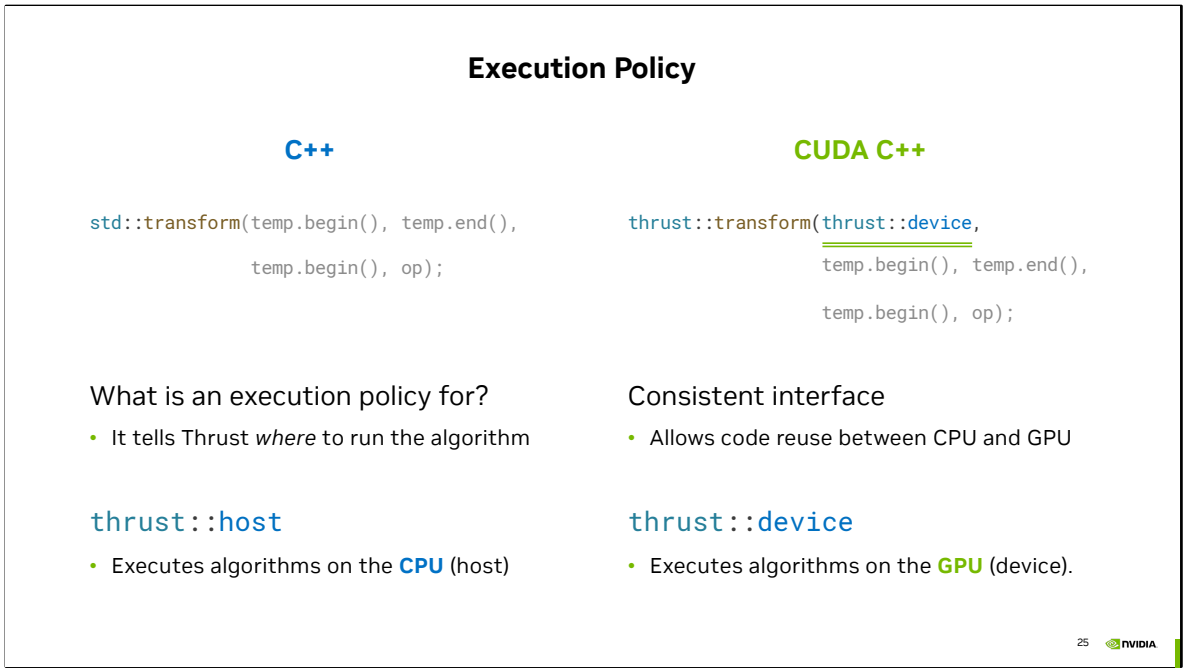



```
thrust::universal_vector<float> temp{ 42, 24, 50 };

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);
```

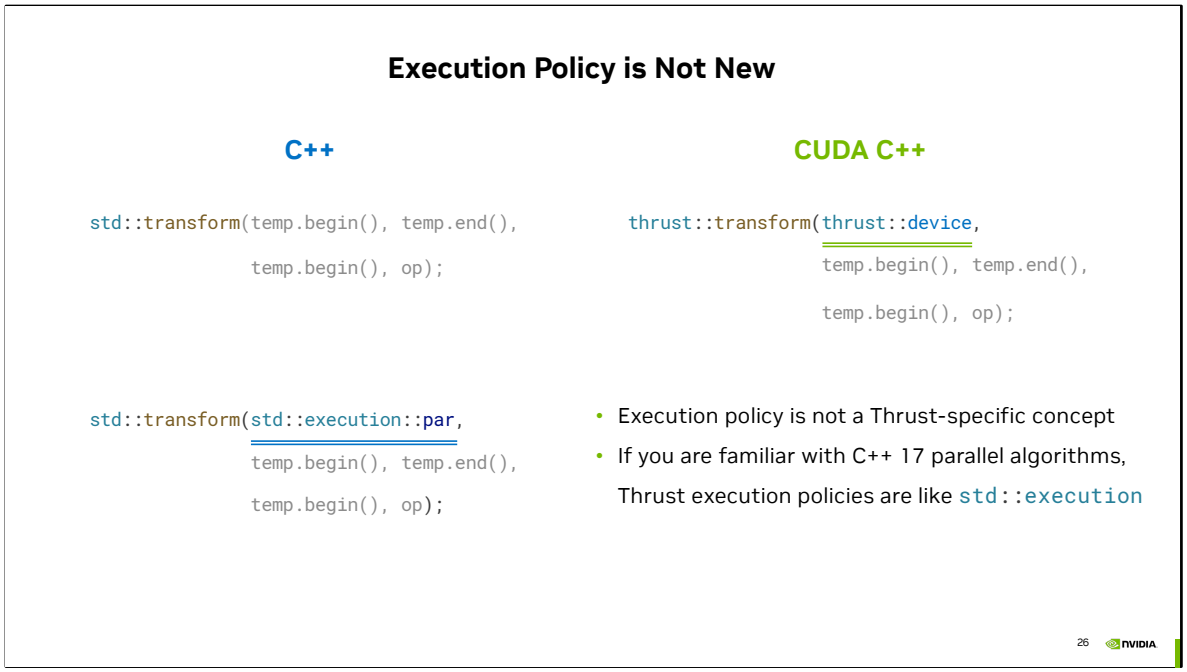
24 NVIDIA

If you want the function to get compiled for both CPU and GPU, you can add __host__ and __device__ specifiers at the same time. And that's exactly what we'll do for the rest of this course.

NVIDIA



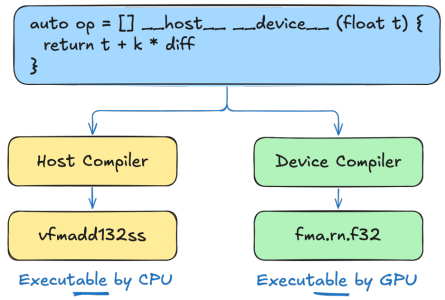
Besides the execution specifiers, we've made another change. In the original code, we used std::transform, and in the GPU version, we use thrust::transform. Aside from the difference in the namespaces, the main change is that we added thrust::device as the first parameter. This parameter is called *execution policy.* It tells Thrust where to run the algorithm. For example: thrust::host runs the algorithm on the CPU, behaving much like std::transform; thrust::device, on the other hand, runs the algorithm on the GPU. Thanks to this consistent interface, you can write your algorithm once and then decide whether it runs on the CPU or GPU. This eliminates the need to maintain two separate code paths and makes switching between CPU and GPU as easy as changing the execution policy.



To be on the same page, execution policies aren't unique to Thrust. Instead, they're a broader idea that even inspired the parallel algorithms introduced in C++17. In both Thrust and C++17, execution policies are provided as first parameter and modify *how* the code runs. But Thrust goes a step further by also specifying *where* the computation happens — on the CPU or GPU.

Execution Policy vs Specifier

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



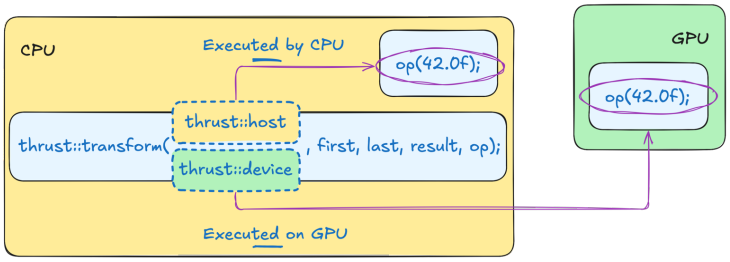
27 NVIDIA

It's important that you understand the difference between the execution specifiers and execution policies, so let's contrast them. When you mark a function with something like __host__ or __device__, you're telling the compiler to generate instructions for CPU, GPU, or both. It doesn't automatically make the function runs there - it just prepares the code for that execution space. So, if you annotate a function with __device__ and call it from CPU, it'll just fail.

NVIDIA

Execution Policy vs Specifier

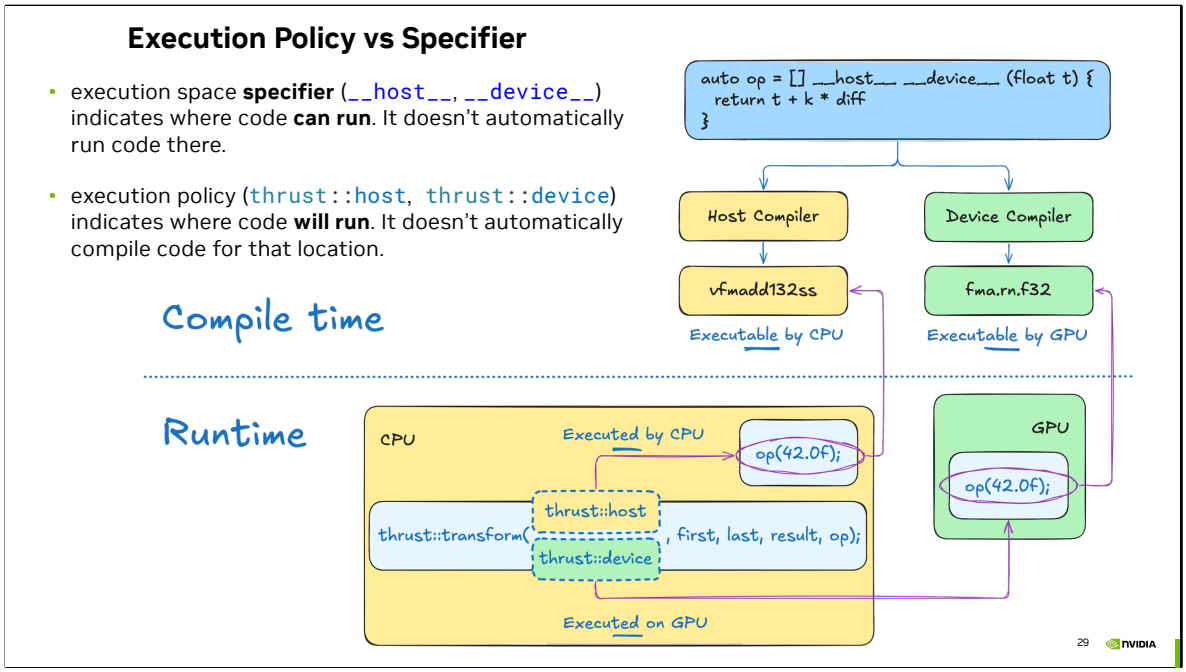
- execution policy (`thrust::host`, `thrust::device`):
  - works at runtime
  - indicates where code **will run**
  - doesn't automatically compile code for that location.



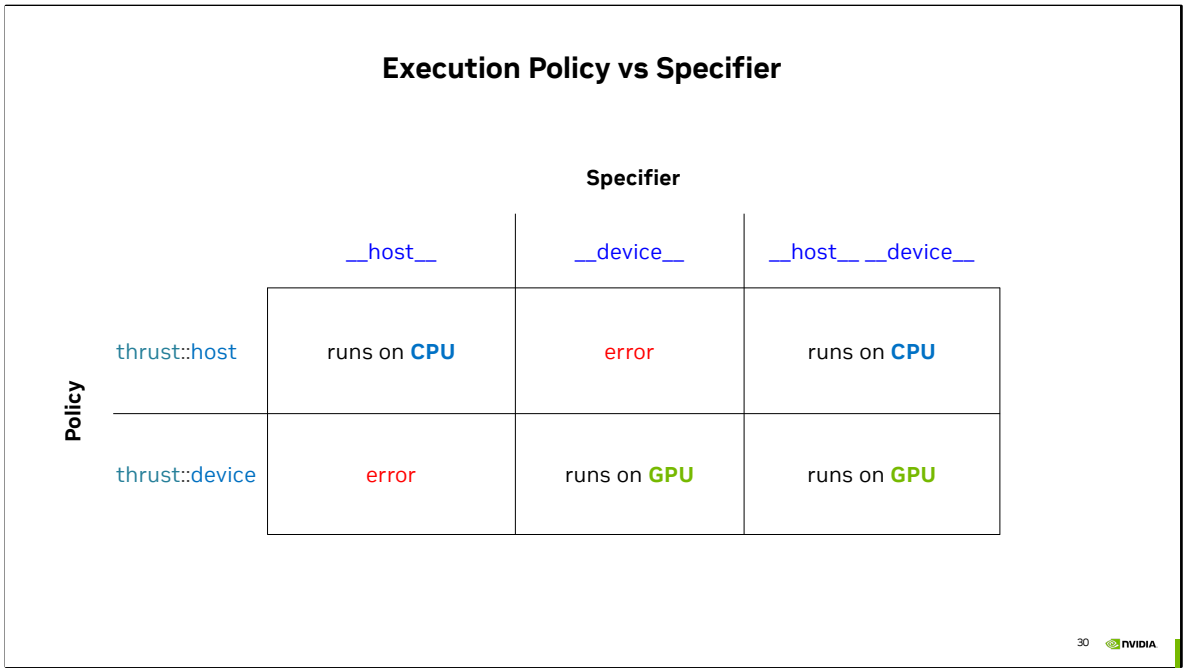
28 NVIDIA

On another hand, an execution policy (like thrust::device or thrust::host). If you pass `thrust::device`, it's equivalent to saying: "Now run this algorithm on the GPU". But this does *not* generate any new CPU or GPU instructions. It simply picks already-compiled version should be executed.

NVIDIA



In short, you need both pieces: 1. **Execution space specifiers** (like __host__ __device__) tell the compiler to generate the required CPU and/or GPU code. 2. **Execution policies** (like thrust::host or thrust::device) decide where that code actually runs. If you don't use the specifier, there's no GPU version compiled. And if you don't use the execution policy, there's no instruction to run on the GPU. They work hand in hand to enable GPU acceleration.



This table shows how the *specifier* (__host__ and __device__) combines with the *policy* (thrust::host or thrust::device) to determine where code runs. __host__ functions are CPU-only, so Thrust fails if you try to run them on the GPU. __device__ function are GPU-only, so Thrust fails if you try to run them on the CPU. Finally, __host__ __device__ functions are compiles for both, and the policy decides whether it actually runs on CPU or GPU.

Exercise: Annotate Execution Spaces

- Replace ??? with CPU or GPU

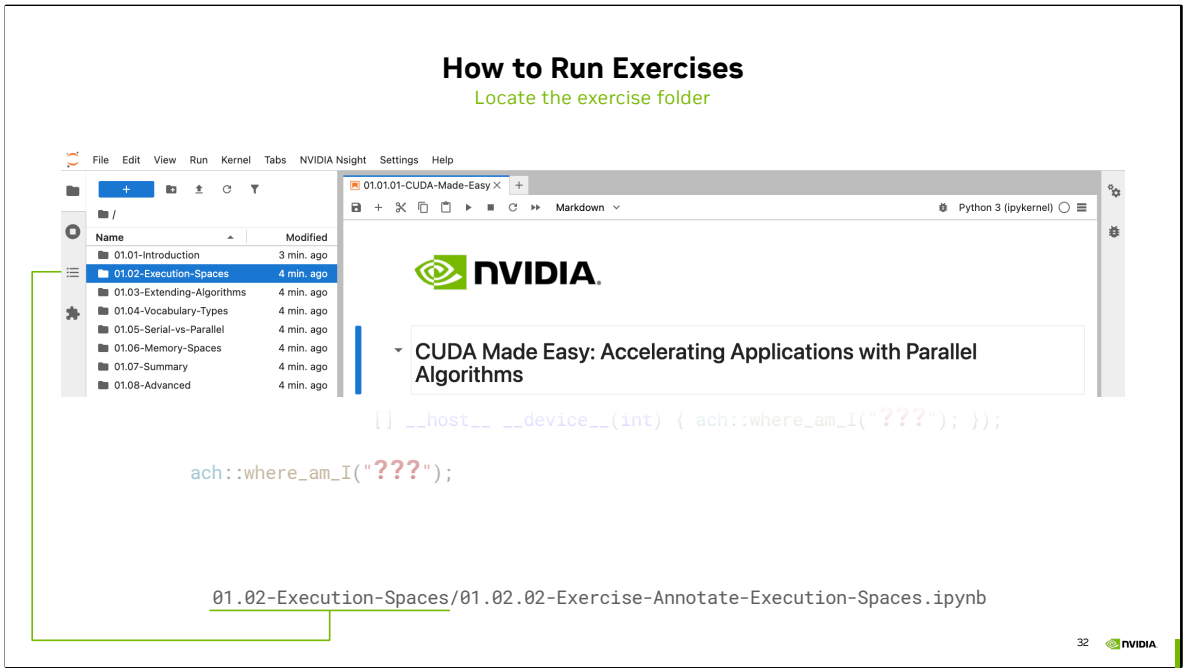
```
ach::where_am_I("???");

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

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

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

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

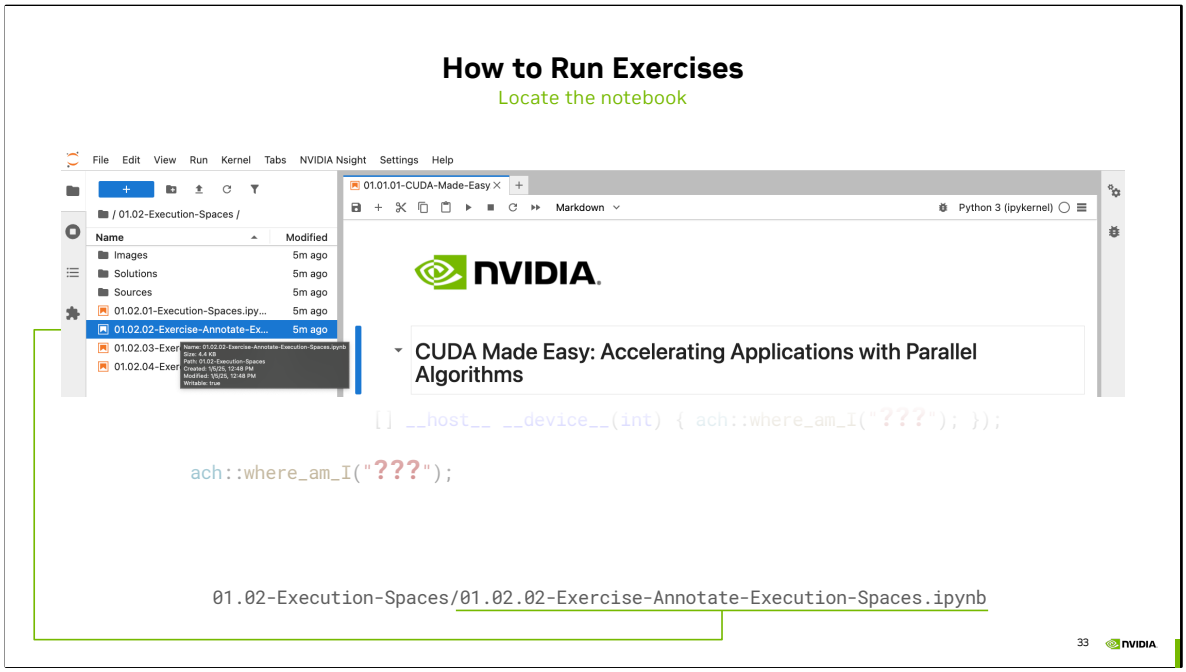
31 NVIDIA

Understanding *where* your code is executed is fundamental to harnessing the power of accelerated computing. In this exercise, you will confirm your understanding of the execution spaces. In the file you see on the slide, you'll have to replace all question marks (?) with either CPU or GPU.

NVIDIA

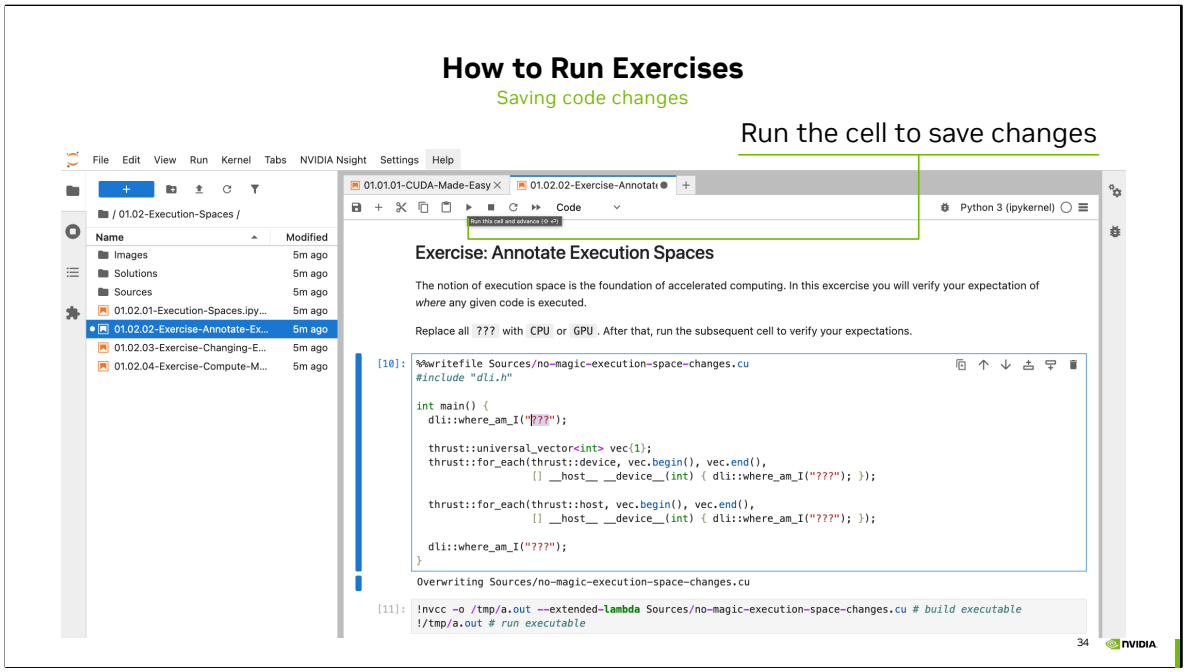


To locate the file, first open the folder that contains it. On the left, you should find 01.02-Execution-Spaces folder.

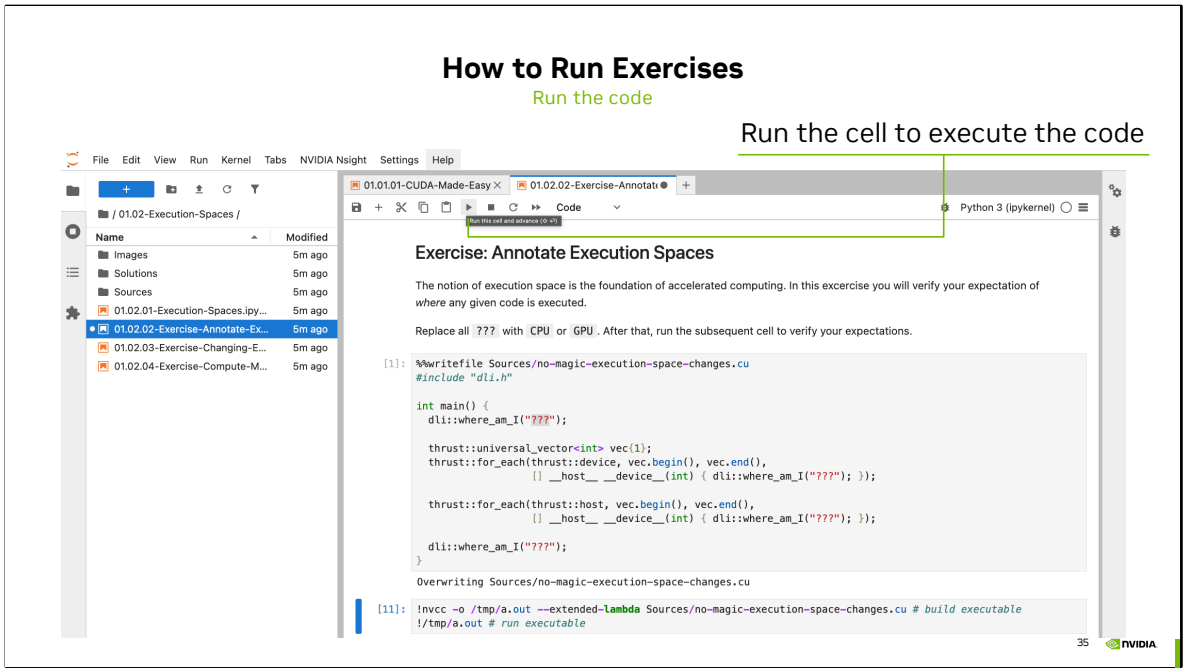


When you click on it, you'll see 01.02.02-Exercise-Annotate-Execution-Spaces. Click on it to proceed to the exercise lab.

NVIDIA

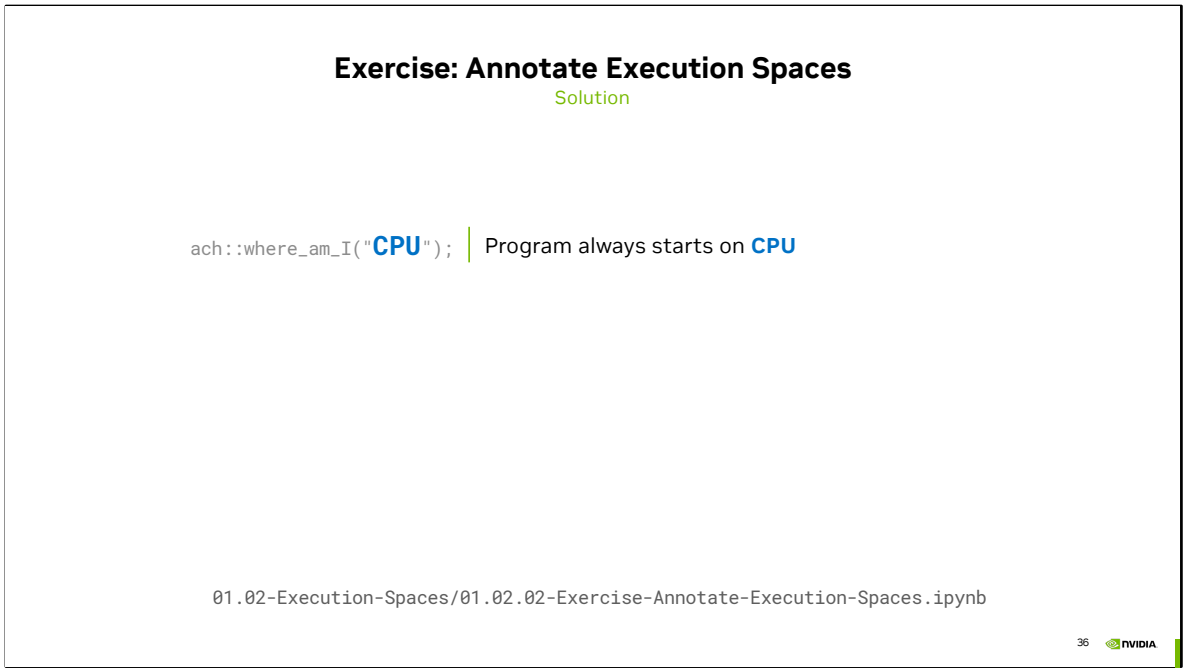


Inside the exercise lab, you can just start editing the code. When you are comfortable with the changes, you can run the cell as shown on the slide to save them.



After that, run the following code cell with NVCC invocation to compile and execute your program.

NVIDIA



Time’s up! Let’s revisit the correct solution. As discussed previously, program always starts on CPU.

Exercise: Annotate Execution Spaces

Solution

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

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

37 NVIDIA

And device execution policy leads to the first lambda being executed on the GPU.

Exercise: Annotate Execution Spaces

Solution

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

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

thrust::for_each(thrust::host, vec.begin(), vec.end(),
                 [] __host__ __device__(int) { ach::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`

Host execution policy leads to algorithm execution on the CPU.

Exercise: Annotate Execution Spaces

Solution

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

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

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

ach::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

And finally, program ends on CPU. I hope this example manages to help you understand that there are no magical execution space changes.

Exercise: Change Execution Space

- 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, ach::execution_space());
});
```

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

40 NVIDIA

Now, let's verify your understanding of execution specifier and policies. Modify the code on the screen so that lambda is executed on GPU.

NVIDIA

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, ach::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, ach::execution_space());
});
```

41 NVIDIA

Time's up! To make this for each algorithm run on the GPU, we have to use thrust::device execution policy instead of thrust::host.

Exercise: Change Execution Space

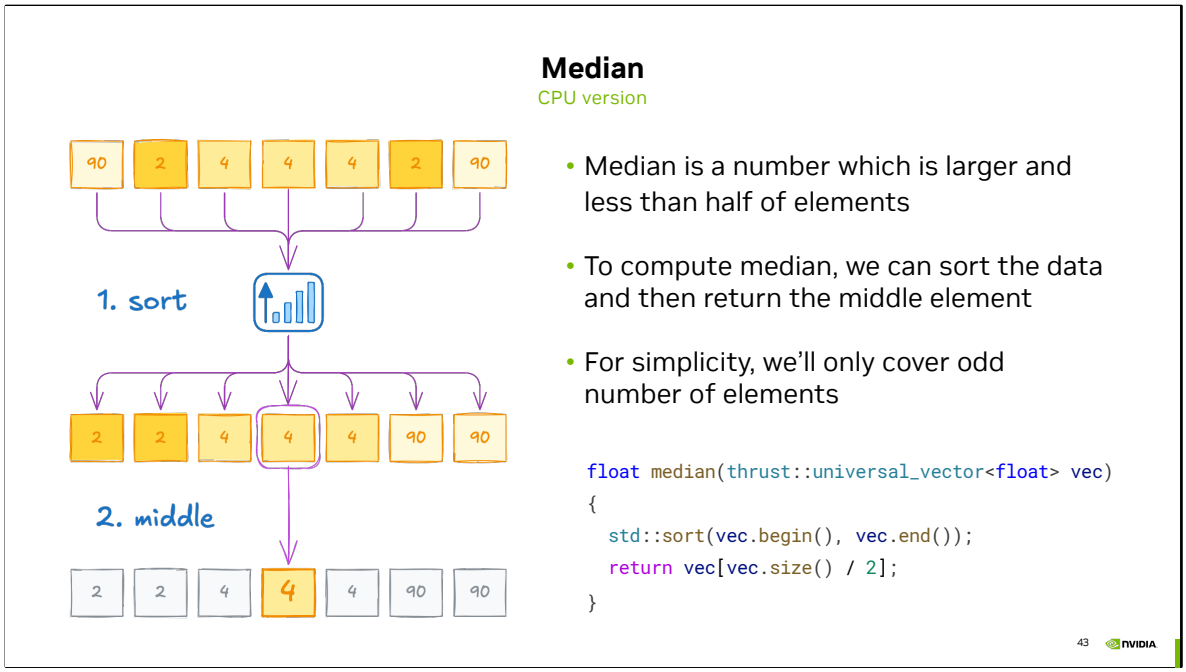
Solution

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


Besides that, we have to change the execution space specifier to `__device__`, so that our lambda function is compiled for the GPU.



Before moving on, let's confirm you understand how to port standard C++ algorithms to the GPU. Suppose we need to track the median temperature in our simulator. To compute median, we can sort the data and then return the middle element. For simplicity, we'll only cover cases when number of elements is odd. And to implement this in C++, it's sufficient to call std::sort and return middle element of the sorted vector.

Exercise: Port to GPU

- 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

44 NVIDIA

In this exercise, you'll have to GPU-accelerate the `median` function that you see on the screen.

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**

45 NVIDIA

Before showing the correct solution, let's highlight a common mistake: trying to mark the median function with __device__. Because `std::sort` is not annotated as device function, this approach won't compile. But even if it did, median is called from the CPU in main, so you can't call a device function directly from host code. Besides that, `std::sort` is serial algorithm, it won't be able to leverage massive parallelism provided by GPUs. What we need is a parallel algorithm, and parallel algorithms are invoked on the CPU.

NVIDIA

Exercise: Port to GPU

Solution

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

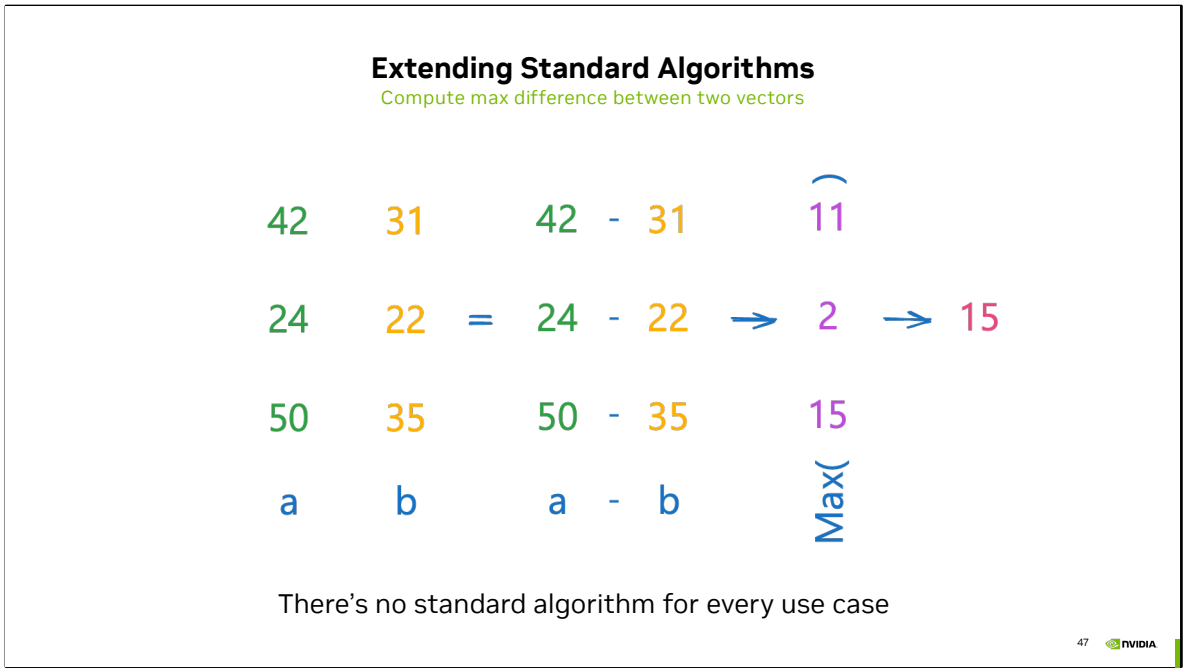
```
float median(...)
{
  thrust::sort(thrust::device, vec.begin(), vec.end());
  return vec[vec.size() / 2];
}
```

GPU-accelerating standard algorithms should be as easy as:

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

46 NVIDIA

That's why the correct solution is to replace std::sort with thrust::sort and use thrust::device as the execution policy. If you completed this exercise correctly, you now know how to adapt many standard C++ algorithms to run on GPUs using Thrust.

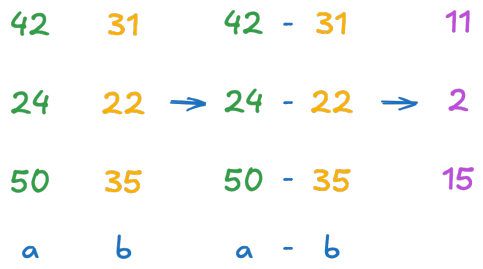


But what if your problem is not a standard one? For instance, let's say we want to compute maximal difference between old and new temperatures. For instance, first object cooled from 42 to 31 degrees, which is an 11 degrees change. But the last one cooled from 50 to 35, which is a difference of 15 degrees. So, the maximal difference is 15 degrees. There's no standard algorithm for computing maximal difference, though. A naïve approach would be to compute each stage of this algorithm as illustrated.

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);
  });
```

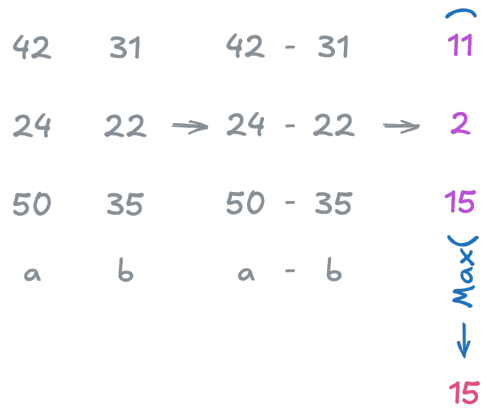


To do this, we start by creating a vector for differences. Then we use the version of thrust::transform that takes **two** input ranges - one for the old temperatures and one for the new temperatures. Our operation (a lambda or function object) now receives **two parameters** (one from each range), computes the absolute difference between them, and writes the result into the differences vector.

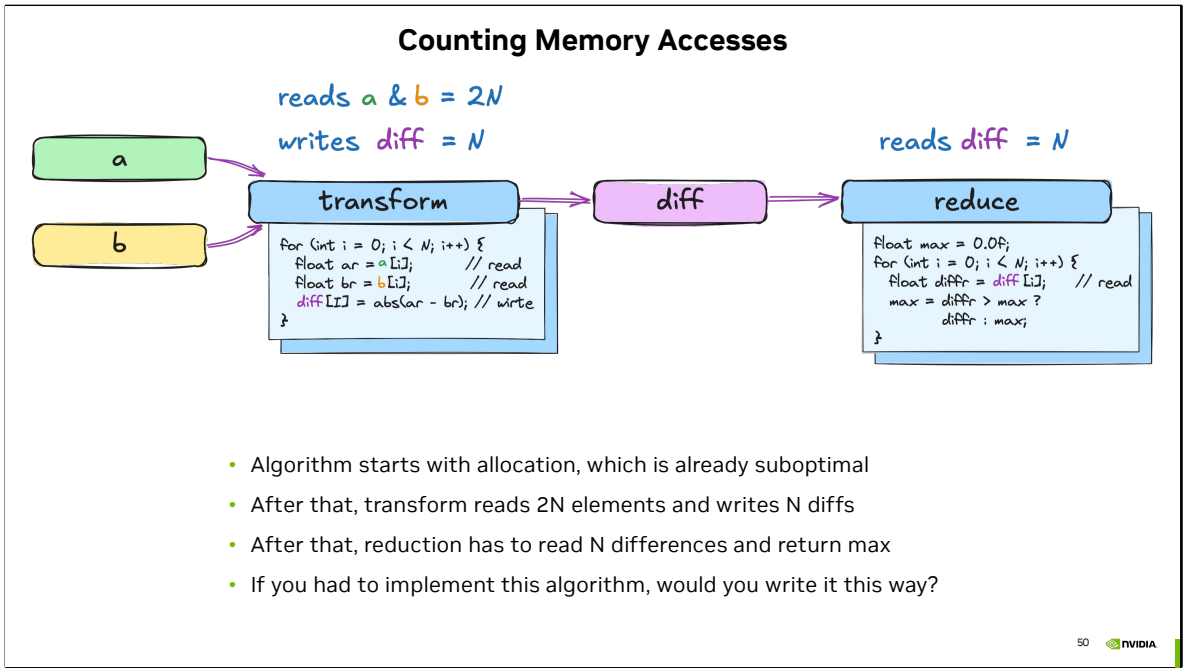
Extending Standard Algorithms

Compute max difference between two vectors

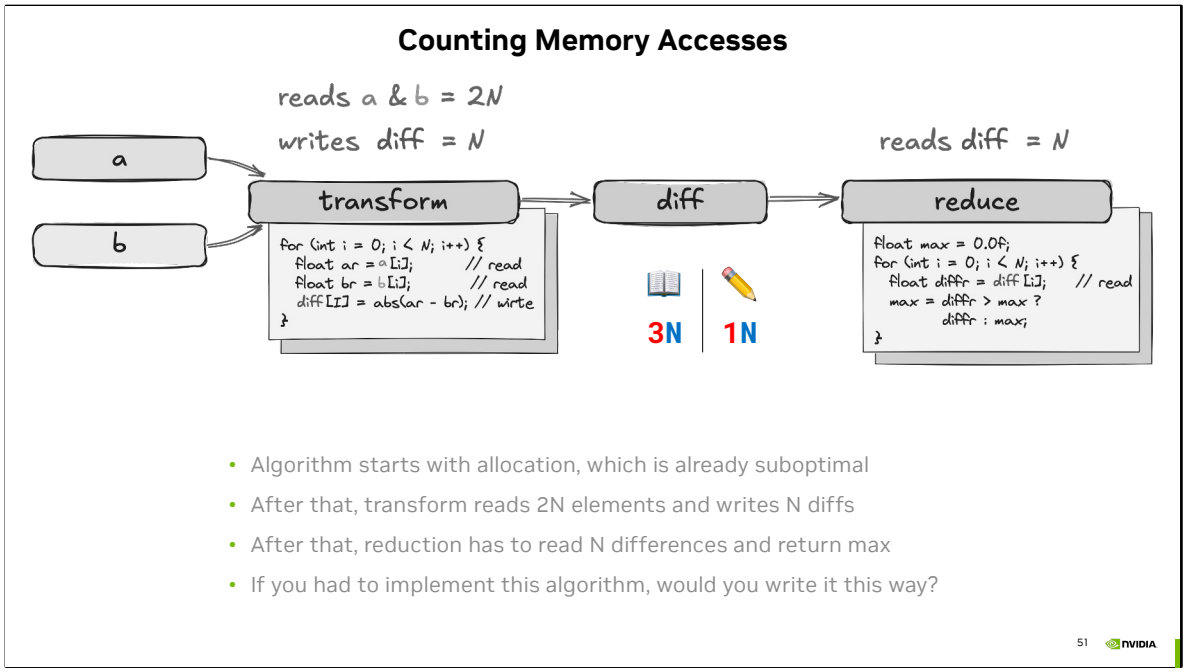
```
thrust::universal_vector<float>
  unnecessarily_materialized_diff(a.size());

// 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>{});
```



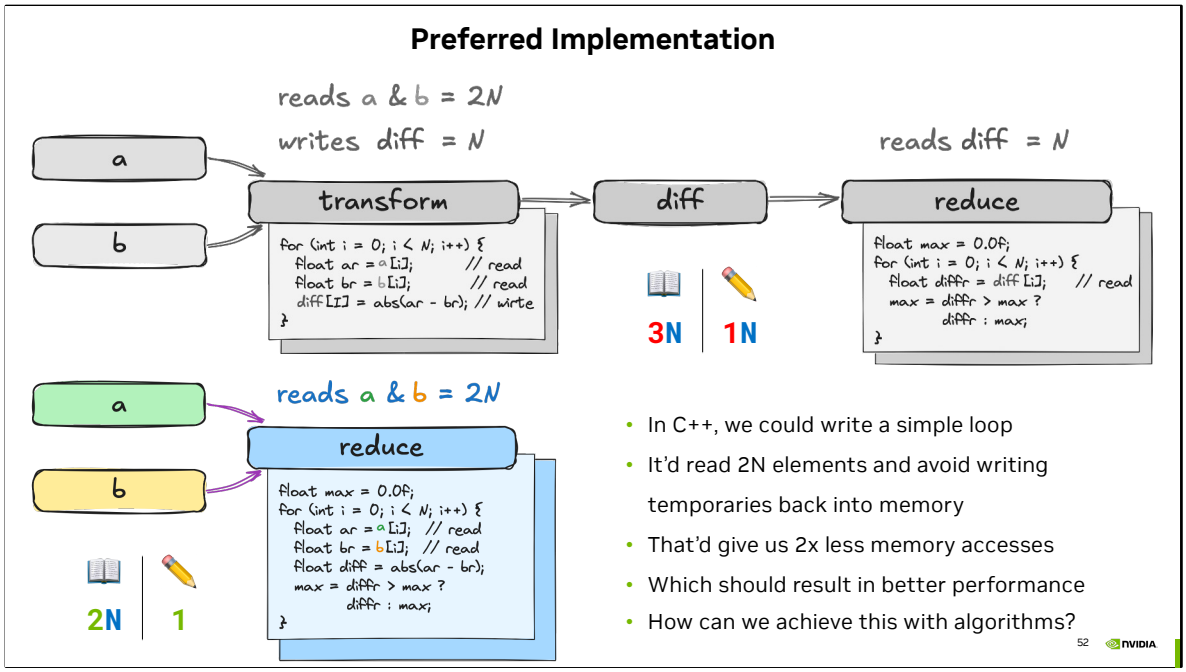
Lastly, we run a max reduction on the differences vector to find the maximum absolute difference. This code will produce the correct result, but is it efficient?



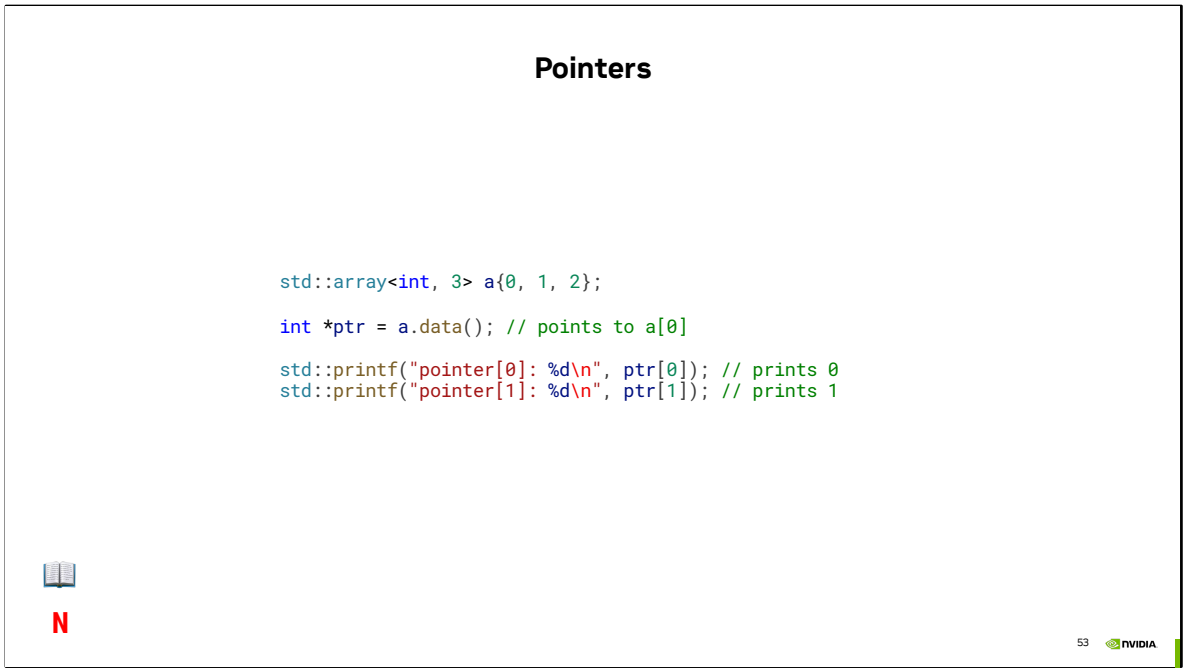
To answer this question, let's recap the key steps of this algorithm. It starts with an allocation, which is already suboptimal. After that, it transforms two input ranges into differences. To build an intuition for this step, you can consider the transformation as a for-loop on the screen. Inside this loop, we read two elements, one from a, and one from b. We then compute the difference and write it back into memory. Overall, this step reads 2N elements and writes N elements back. The subsequent reduction step, can be represented as another for loop. Here it reads one element from difference array, checks if the newly read value is larger than the current max value, and updates the current max value if it's the case.



Overall, this approach leads to 3N reads, where N is the size of the input sequence. It also write back N elements. The question is, if you had to implement this algorithm with just for loops, would you write it this way?



Personally, I'd implement this with a single reduction loop: for each element in a and b, compute the absolute difference on the fly and compare it to the current maximum. This way, we only read 2N elements once, and we avoid any extra memory allocations. Since this approach leads to 2x less memory accesses, I'd expect it to be much faster. The next question is: how do we get a parallel algorithm to implement this on-the fly computation of absolute differences? And which parallel algorithm should we start with? Well, at least this part is straightforward - our loop already looks just like a standard reduction, except that it's reducing absolute differences instead of the raw values. So, it has to be `thrust::reduce`. But how do we embed this logic of computing absolute differences in it? The answer is iterators. If you're not familiar with the concept, don't worry - we'll quickly recap it now.

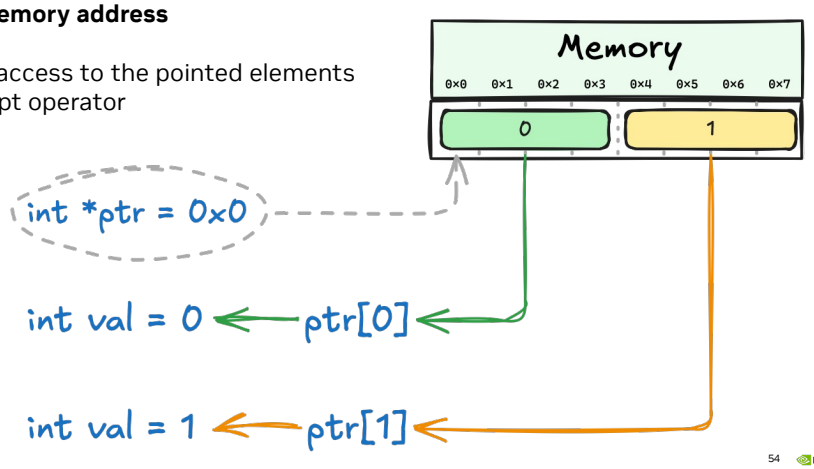


And it's easiest to understand iterators by looking at their simplest form - pointers. On the slide you can see a simple program that allocates an array with 3 elements. Then we initialize a pointer with address of the first element. We can then use square brackets, also called subscript operator, to reference underlying integer sequence.

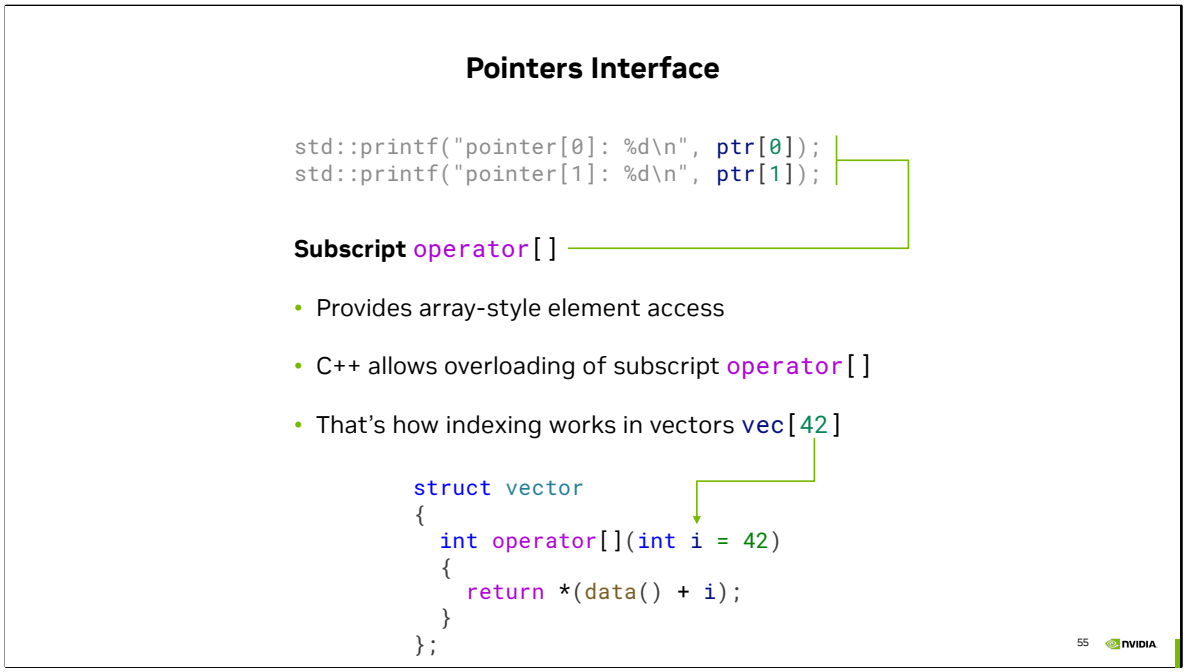
Pointers

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



Here's a very simple mental model for pointers - they store a physical memory address. Besides that, pointers provide an array-style interface for accessing the data they point to. So, if I write pointer at index zero, I get the first element of the sequence; if I write pointer at index one, that moves me to the second element in memory, and so on.



But the subscript operator isn't limited to pointers and arrays. If you think about how we access elements in thrust::vector or std::vector, we use the same square bracket notation. Why does that work? Because in C++, we can define the subscript operator just like any other function. In other words, when you write 'vector at index 42', 42 is passed as a parameter to the operator[] function inside the vector class. If it's hard to treat subscript operator as normal function, try replacing square brackets with parenthesis to recognize it as a normal function call. Anyways, this flexibility allows us to give many objects pointer-like behavior, which brings us to iterators.

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;
  }

};
```

56 NVIDIA

Since we can put any code in the subscript operator, we don't have to limit ourselves to the concept of physical memory at all. For example, imagine a type whose subscript operator just returns whatever index you pass in. Pass it 42, you get 42. That might seem pointless at first but trust me - by the end of this course, this little trick will become one of your best friends!

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

57 NVIDIA

So now let's compare the pointer-based code on the left with our custom type on the right. When we pass in an index of zero, both print zero. When we pass in an index of one, both print one, and so on. Functionally, there's no difference. But because a pointer is tied to actual physical memory, every time you use the subscript operator, you're performing a memory access. That means if you read N elements, you'll have N memory accesses. Our custom type, on the other hand, simply returns the index you pass in, with no memory access at all. In theory, that can give much better performance. And from the algorithm's point of view, notice how both interfaces look exactly the same: we just use square brackets. Whether those brackets trigger a memory read or simply return the index doesn't matter for the rest of the code - and that's a key concept behind how iterators work.

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

```
counting_iterator it;

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

📖

```
0
```

Mental Model:

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

Iterators:

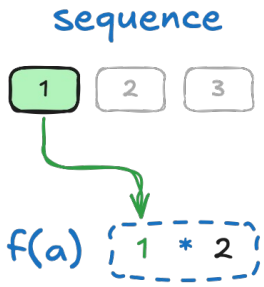
- Lead to same functional behavior
- Reduce memory footprint
- Reduce memory traffic which improves performance

58 NVIDIA

You can think of iterators as a generalization of pointers. They provide the same interface, think subscript operator, increment, etc., but they're not limited to actual memory accesses. The type we just explored is called a counting iterator: it presents an infinite sequence of integers without allocating a single byte. I'm showing you its code just to help build a mental model, but you won't ever need to write one yourself. It's the concept that's important.

NVIDIA

Simple Transform Iterator



Mental Model:

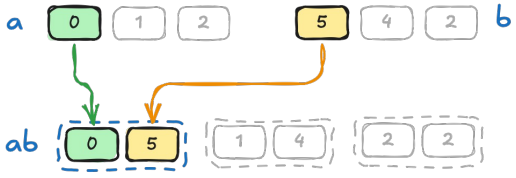
```
struct transform_iterator
{
  int *a;

  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
```


The counting iterator we just saw doesn't make any memory accesses, but not every iterator works that way. For example, imagine a 'transform iterator' that references an underlying pointer. Every time you call its subscript operator, instead of returning the original element directly, it might multiply that element by two before giving it back. From the outside, it looks like the entire sequence was scaled - if you look at index one, you get two, even though the underlying data still stores one. Notice that the result of the multiplication isn't stored anywhere; it's computed on the fly each time you access an element.

Simple Zip Iterator



Mental Model:

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

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

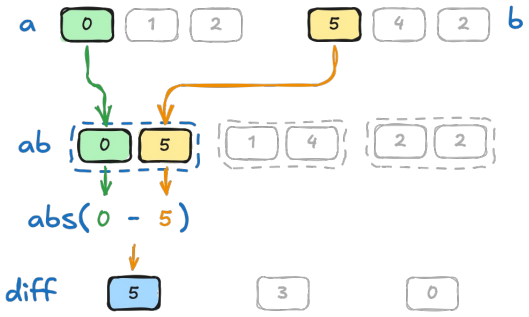
```
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> b{ 5, 4, 2 };

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

We can take this even further by implementing a ‘zip iterator’ that references two sequences at once. Its subscript operator returns a tuple containing one element from each sequence. So, if we access the iterator at index zero, we might get (0, 5), which corresponds exactly to a[0] and b[0], as shown on the slide.

Combining Input Iterators



You can nest iterators!

📖

2N

Mental Model:

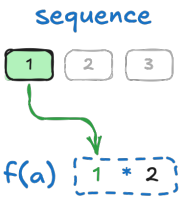
```
struct transform_iterator
{
  zip_iterator zip;

  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]);
```

61 NVIDIA

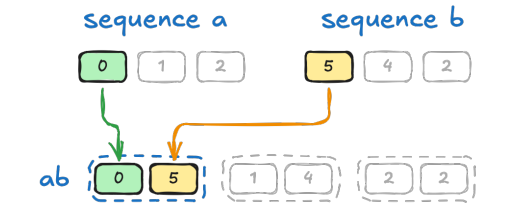
The real power of iterators shows up when we start combining them. For example, imagine a transform iterator that uses a zip iterator inside. Each time you call its subscript operator with index `i`, it first fetches two values - one from sequence a, one from sequence b - then computes their absolute difference and returns it. That's exactly what we needed for our maximum difference algorithm! If we pass this 'transformed zip iterator' to a reduction algorithm, it'll treat it just like a regular pointer and read values as usual, but under the hood, every access fetches two elements, computes the absolute difference, and hands it over. All of that without ever writing those intermediate differences back to memory.

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)
```

62 NVIDIA

In reality, production-grade iterator implementations are much more complex than the simplified examples we've shown on these slides. Luckily, you don't have to build them yourself. Thrust already provides a variety of ready-to-use iterators, including counting, zip, and transform iterators. Let's take a look at how to work with those in practice.

NVIDIA

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
```

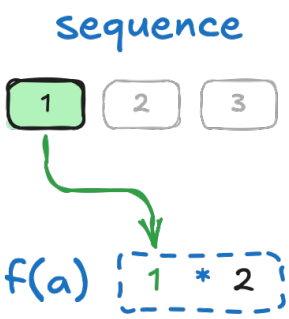
63 NVIDIA

`thrust::make_counting_iterator` takes an integer that tells it where to start counting. For example, in the code on the slide, we pass it into `thrust::for_each` to print out the first 42 elements. If we create a counting iterator that starts at 1, the output will be 1, 2, 3, and so on, up to 42.

NVIDIA

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
```

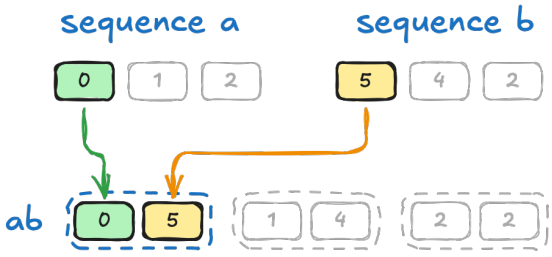


64 NVIDIA

`thrust::make_transform_iterator` takes two parameters: 1. an iterator pointing to your underlying data, and 2. a transformation function that does something to each element. For example, on the slide we have a Thrust vector containing a simple sequence like 1, 2, 3, and so on. We then pass the beginning of that vector as the first parameter to `make_transform_iterator`, and as the second parameter, we provide a function that doubles any value it receives. That means when we iterate over this transform iterator, each element is automatically multiplied by two on the fly. If our underlying sequence was 1, 2, 3, 4..., then the transform iterator produces 2, 4, 6, 8, and so on. Notice that we never actually store these doubled values in memory; they're generated each time we access an element through the iterator.

NVIDIA

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)
```

65 NVIDIA

Now let's take a look at an example using `thrust::make_zip_iterator`. We have two thrust vectors, a and b. The first one holds {0, 1, 2}, and the second holds {5, 4, 2}. We then call `thrust::make_zip_iterator(a.begin(), b.begin())` to create an iterator that pairs up elements from both vectors: the element at index 0 in a is zipped with the element at index 0 in b, and so on. That gives us sequence of pairs like (0, 5), (1, 4), (2, 2).

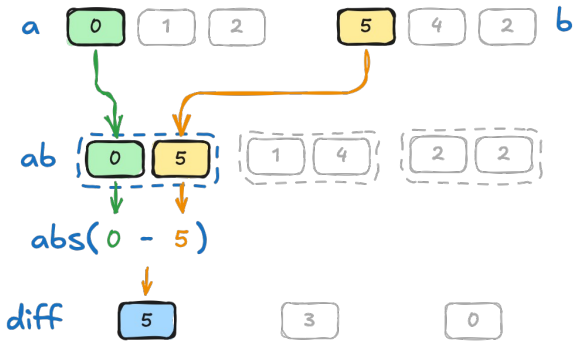
NVIDIA

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:

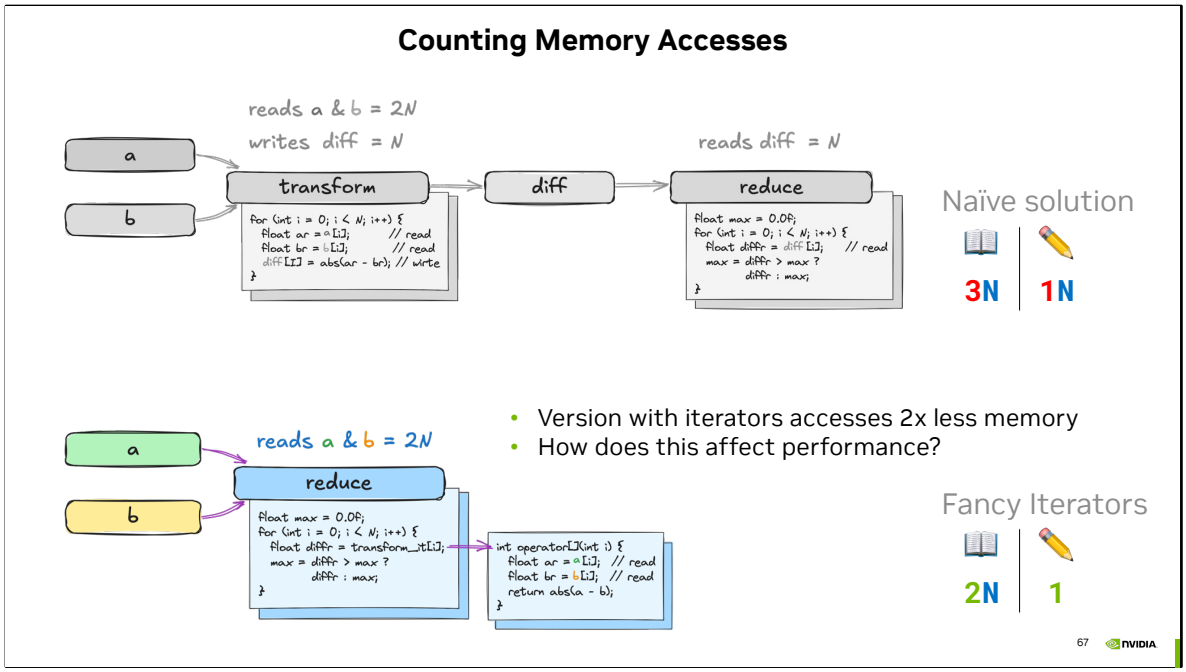
- `zip_it[i]` returns tuple
- `transform_it[i]` returns diff



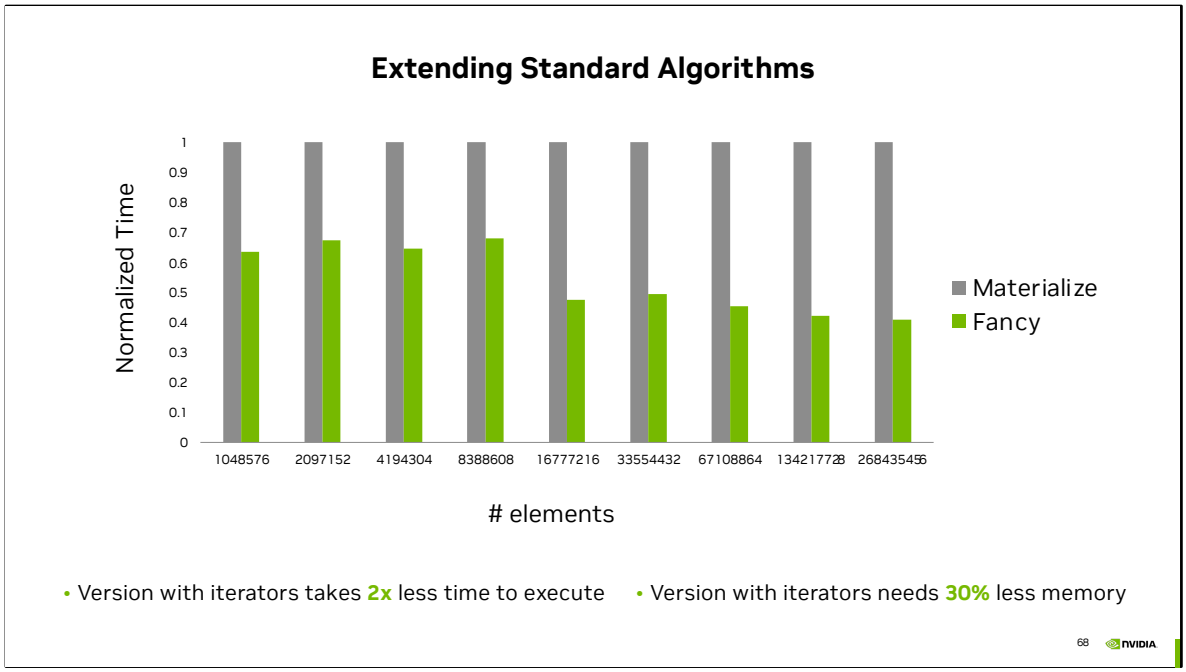
66 NVIDIA

As we discussed, our plan for the maximum difference problem is to combine a zip iterator with a transform iterator. In this code, we do exactly that. First, we use `make_zip_iterator on a.begin() and b.begin()`. That gives us an iterator which, at each index, returns a tuple of two floats - one from a and one from b. Next, we pass that zip iterator into `make_transform_iterator`, along with a lambda function. Because the zip iterator returns a tuple of floats, that's exactly what the lambda function should have as a parameter. For each tuple, we extract the two values using thrust::get, compute the absolute difference, and return that result. So, for the first elements in a and b, if the tuple is (0, 5), the transform iterator returns abs(0 - 5), which is 5.

NVIDIA



From the reduction's perspective, it's just reading a single float each time. But behind the scenes, our transform iterator grabs two floats - one from a and one from b vectors - calculates the difference on the fly, and returns that result, without any extra reads or writes. So, in total, we only need 2N memory accesses, which is exactly the improvement we aimed for. Now let's see if this reduced memory traffic actually improves performance.



On this slide, we have normalized elapsed times for different vector sizes. Remember, lower is better. You can see that using fancy iterators is about twice as fast as storing the temporary differences in memory.
And this comparison doesn't even include the cost of allocating space for those temporary arrays. If we factor in the allocation overhead, the speedup could climb to around 70 times! But performance isn't the only benefit. Fancy iterators also drastically reduce memory usage. For instance, if your GPU has 8GB of memory and each vector takes up 4GB, then you've essentially used up all available space by just storing a and b. In that scenario, a materializing approach—which needs extra storage for the difference vector—would fail. Meanwhile, the fancy-iterator-based version requires no extra space and would work just fine.

Exercise: Computing Variance

- 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();
}
```

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

Most of the parallel algorithms we've seen so far are very challenging to implement efficiently. That's why it's crucial for you to understand how to adopt these accelerated libraries to your unique use cases without having to re-implement them.Let's pause and verify your understanding of how to extend parallel algorithms to your unique use cases.This time we want to calculate the variance of a temperature dataset, which requires us to compute the squared difference of each element from the mean. A naive approach would create a whole new buffer in memory to store these squared differences, which wastes both time and space. Instead, you already know how to handle this using 'fancy iterators' - we'll compute squared differences on the fly, feeding them straight into the reduction without ever creating a separate array. In the code snippet, you'll see a placeholder for squared_differences. Your task is to replace that placeholder with a transform iterator that, when dereferenced, returns squared difference of incoming value with the mean.

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();
}
```

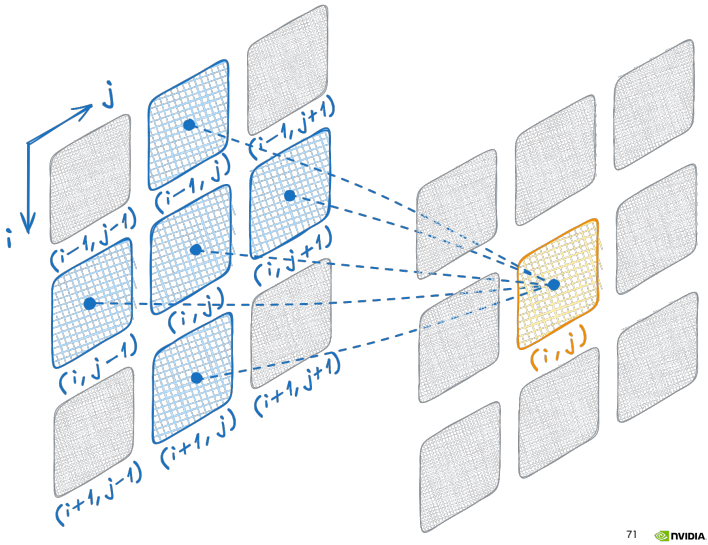
70 NVIDIA

The solution is straightforward: we use a Thrust transform iterator. This time, we capture the mean in our lambda and compute each element's squared difference on the fly. By now, you should have a clear idea of how to apply fancy iterators and parallel algorithms in practice. So, it's a good time to complicate our simulator even further!

NVIDIA

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



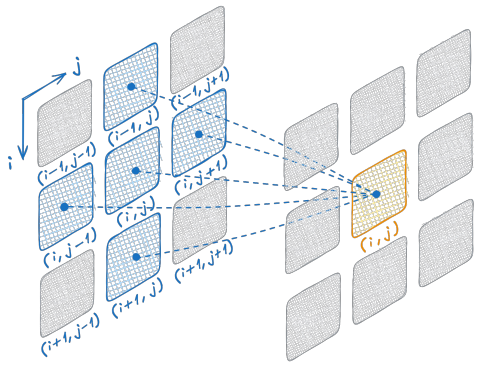
71 NVIDIA

So far, we've looked at simple temperature simulations where each cell or object evolved on its own, independent of the others. Of course, that isn't very realistic. Now that we're comfortable with fancy iterators, parallel algorithms, and execution spaces, let's tackle a slightly more complex scenario: the 2D heat equation. On the screen, you can see the computational stencil that we'll use. Each cell's temperature in the next time step depends not only on its current temperature, but also on its four neighbors - left, right, top, and bottom. Let's see how we can implement this scheme.

NVIDIA

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);
```

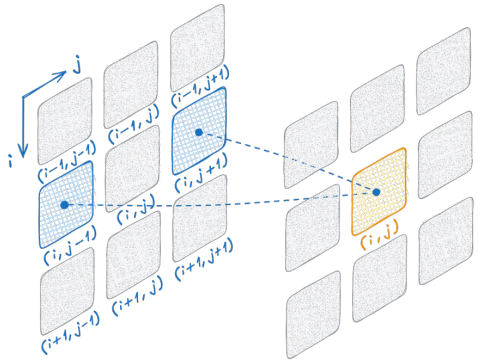
72 NVIDIA

In code, the 2D heat equation might look something like this. Don't worry about the specific math or constants here. Instead, try focusing on the data access pattern.

NVIDIA

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);
```

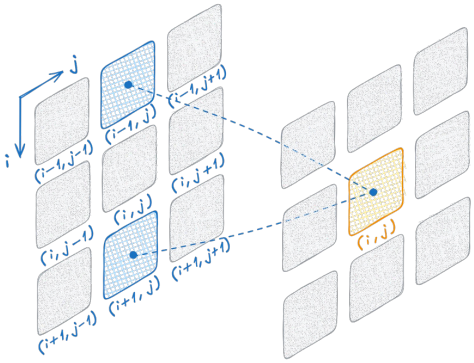
73 NVIDIA

First, we have to read temperatures of the left and right neighbors of a given cell. A 2D array is typically laid out in row-major order, meaning each row is placed sequentially in memory. To find a flattened 1D index of a 2D row / column index, we can multiply row index by the width of our grid and then add column index. Here: row * width jumps us to the start of the correct row (because each row has width elements); column then offsets us within that row, taking us to the specific column. Using this same pattern, if we want the left neighbor of a given cell, we just subtract one from the column index. For the right neighbor, we add one instead.

NVIDIA

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);
```

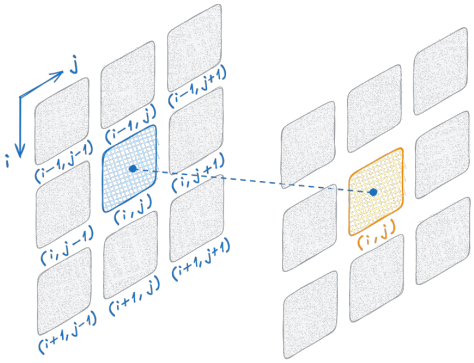
74 NVIDIA

Similarly, we compute index of top and bottom neighbors by subtracting and adding one to current row index.

NVIDIA

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);
```

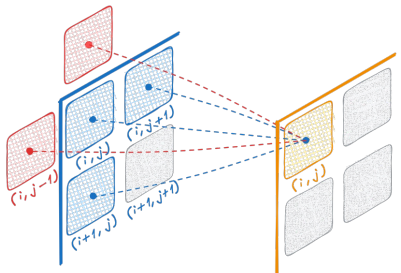
75 NVIDIA

Besides that, we incorporate previous temperature of a given cell. This code works fine in most of the cases, but there's one issue.

NVIDIA

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];
}
```

76 NVIDIA

Real grids aren't infinite, so cells along the edges don't have all four neighbors. If we tried to compute a left neighbor for the leftmost cell, for example, we'd go out of bounds. To prevent that, we need to handle boundary conditions. In this course, we're using 'constant' boundary conditions. This means that we don't update the temperature for boundary cells - they keep their original values. That's why the code has an if statement checking whether row and column are strictly inside the grid. If a cell is on the edge, we skip the update and return its current value.

NVIDIA

Implementing Stencil Pattern

We can now:

- capture pointer to previous temperatures,
- and transform cell indices to new values

```cpp
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

77 NVIDIA

The next question is how we actually launch this computation on our 2D data. If you think about it, each cell in our grid corresponds to a single ID in a flattened, one-dimensional space. So, we can use a `thrust::make_counting_iterator` to generate all possible cell IDs - one for each cell - and then transform those IDs into updated temperature values. But we still need to figure out which row and column these flat IDs corresponds to. That's where the modulo and division come in. By dividing cell index by module of matrix width we get a column index, and by dividing the cell index by the matrix width, we get the row index. With this, we map each flat index to its correct 2D position. After that, the rest of the code is the same boundary-check logic that we used before. Essentially, we're just transforming cell IDs into next-state temperatures, without needing an explicit loop over rows and columns.

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;
```

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()`

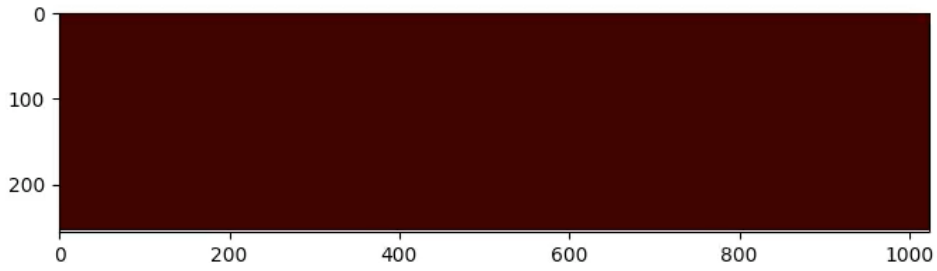
- Returns a *typed iterator*
- You can convert it to a raw pointer with `thrust::raw_pointer_cast`

NVIDIA

Notice that we're capturing the pointer to our temperature data in a slightly unusual way. In standard C++, calling .data() on a std::vector returns a raw pointer. But with Thrust containers, .data() returns a typed iterator instead of a raw pointer. Thrust needs that type information for various reasons, which we'll explore later. For now, the main point is that you can convert this typed iterator to a regular C++ pointer by using thrust::raw_pointer_cast. This gives us the in_ptr we can capture in the lambda.

Implementing Stencil Pattern

Simulation results



Let's simulate an empty oven:

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

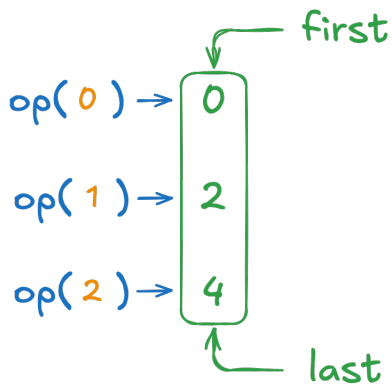
79 NVIDIA

We can start by initializing the entire grid to a low, uniform temperature and then set some higher temperature values at the top and bottom boundaries. If we run our heat-equation transformation with those initial conditions, we get results that look a bit like an empty oven, heating from the top and bottom. Over time, you can see the heat spreading into the cooler regions, all according to the stencil computations we just set up.

NVIDIA

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)

80 NVIDIA

An important thing to remember is that Thrust isn't limited to algorithms that match the exact signatures of the C++ Standard Library. If there's a common pattern that can be accelerated on the GPU, Thrust often provides a specialized algorithm for it. A good example is thrust::tabulate. Under the hood, it's really just transforming a counting iterator - something we've seen is quite common - so Thrust gives it a dedicated name and interface. The call looks just like thrust::transform, except there's no input range to pass in, because the input is implicitly generated by tabulation. Besides being more convenient, specialized algorithms like this are often more optimized. So, if you spot an algorithm whose name and purpose match what you need to do, it's a good idea to use it.

NVIDIA

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

81 NVIDIA

So, let's take the next step and replace thrust::transform with thrust::tabulate. Notice we don't need to create a counting iterator anymore - tabulate generates the indices internally. Apart from that, the rest of the code stays exactly the same, so it's a straightforward swap. This makes our intent clearer and code easier to read.

NVIDIA

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`

82 NVIDIA

We'll likely need to convert from a flat 1D index to 2D coordinates in more than one place, so it's best to follow the Don't Repeat Yourself principle and put that logic in a separate function. That's what `row_col` function does: it returns a std::pair with the row and column calculated by dividing and taking a remainder against matrix width. We've marked this function as __host__ __device__ so it can be called on both the CPU and GPU. Then, in our lambda, we can use structured bindings to unpack the pair directly into row and column variables. This keeps our code clean, makes it easier to maintain, and avoids scattering the same formula in multiple places.

NVIDIA

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?

83 NVIDIA

Unfortunately, the C++ Standard Library doesn't know anything about CUDA, so functions like `std::make_pair` aren't marked as __host__ __device__. Calling them inside device code triggers a compilation error like the one you see on the screen: 'calling host function from a host-device function.' So, what are our options here? Do we have to re-implement every standard library type just to make it device-friendly?

NVIDIA

Vocabulary Types in libcu++

Compound Types

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

Optional and Alternatives

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

Math

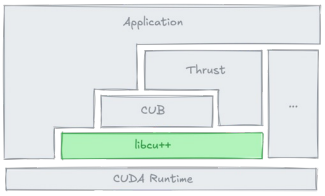
- `cuda::std::complex`
- `cuda::std::mdspan`
- `...`

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

84 NVIDIA

Fortunately, you don't have to do that. Recall from the stack diagram that Thrust is built on top of libcu++, a core library that provides host-device-friendly implementations of many standard C++ vocabulary types. For instance, it provides such types as pair, tuple, optional, variant, complex, and atomic. Not only do these types carry the proper __host__ __device__ annotations, but they're also optimized for GPU usage. That means you can treat them as drop-in replacements for the usual C++ types, and they'll compile and run correctly on both the CPU and the GPU. Let's see how to swap out std::pair for the libcu++ equivalent.

NVIDIA

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**

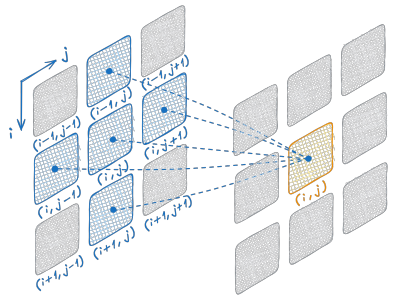
85 NVIDIA

The actual change couldn't be simpler: we just prefix `cuda::` to `std::pair`, making it `cuda::std::pair`, and now our code compiles and runs without issues. The same pattern applies to other standard library types - just pull them in through `cuda::std` instead of std, and you'll have GPU-friendly, optimized equivalents.

NVIDIA

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];
```

86 NVIDIA

cuda::std::pair isn't the only type that can clean up our code. Right now, we're still manually converting between 2D row/column coordinates and 1D indices. That approach works, but it's easy to make off-by-one mistakes and other indexing errors. It's also not very expressive - you have to remember the exact formula for flattening coordinates every time.

NVIDIA

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

87 NVIDIA

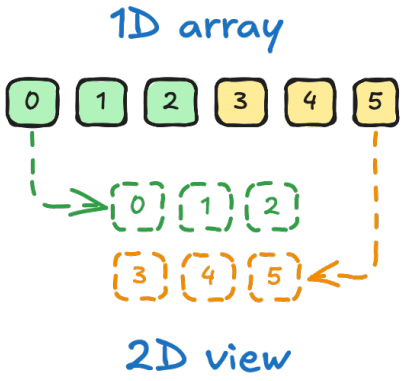
It might not be too bad when we're just flattening two dimensions, but as soon as you go beyond that, things can get ugly quickly. Take a look at the snippet on the slide: it flattens a five-dimensional set of coordinates into a single index. This is actual production code I've come across. It's easy to make a mistake in code like this - even the comment above has to remind you what the developer really intended to write! What if I told you there's a standard C++ type that can make this comment **actually** compile.

NVIDIA

mdspan

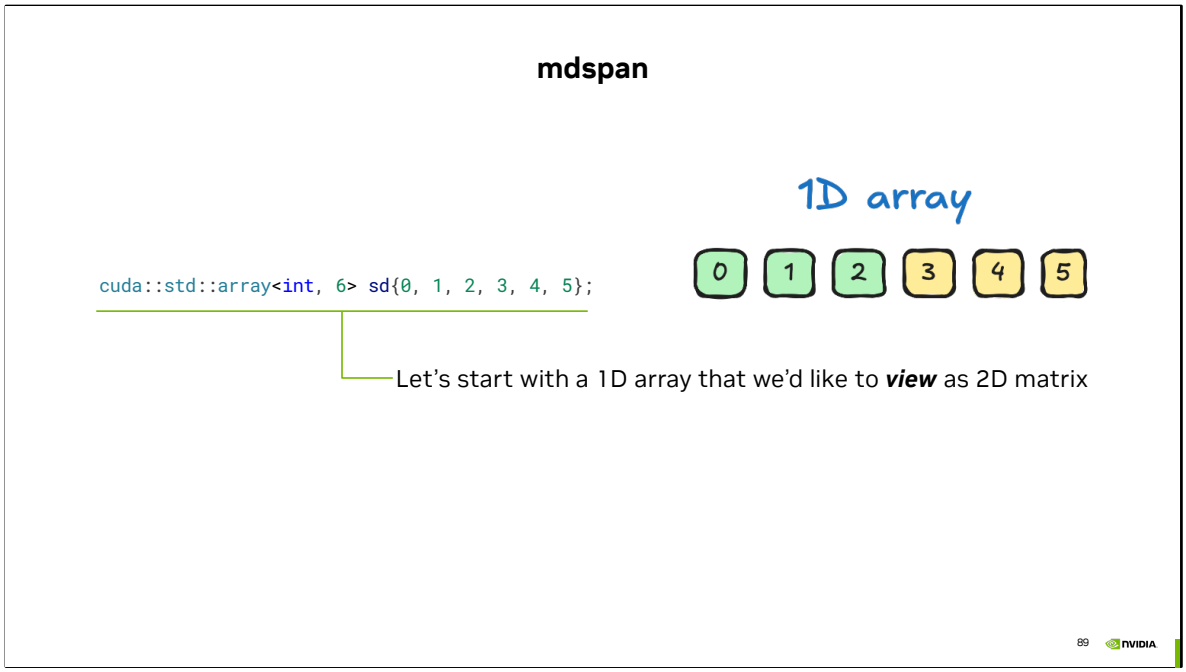
`cuda::std::mdspan`

- 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



88 NVIDIA

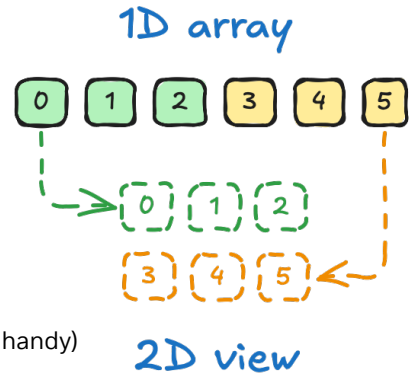
cuda::std::mdspan is a **non-owning, multi-dimensional view** built on top of a flat data array. Think of it like a lightweight wrapper around your memory: you give mdspan the dimensions (for example, height and width), and it automatically handles the offset calculations. Because it's just a view, there's no extra memory usage or hidden overhead - it's practically zero cost.
mdspan results is cleaner and. safer code, which in turn leads to fewer bugs. Instead of scattering row-major index arithmetic everywhere, you can simply provide mdspan row and column for a 2D layout or extend that idea to higher dimensions. The image on the slide shows how a 1D array of six elements can be treated as a 3 by 2 grid through mdspan. Let's take a look at the actual usage of this type.



Because mdspan doesn't own memory, you need an underlying buffer that holds the actual data. In this example, I've created a cuda::std::array<int, 6> containing six elements: 0, 1, 2, 3, 4, and 5.
mdspan will then view these values in whichever multi-dimensional shape we specify, but the data itself remains right here in our array.

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

90 NVIDIA

Let's take a look at how you can construct an mdspan based on this array. For this case, we have to pass mdspan three things. First, we need a raw pointer to the data. Here we use array data() member function that returns a raw pointer. But if you have to construct mdspan on top of Thrust container, remember to use raw pointer cast. Besides the raw pointer, we'll have to provide number of rows (2) and column (3). That's all mdspan needs to treat our 1D array as a 2-by-3 matrix.

NVIDIA

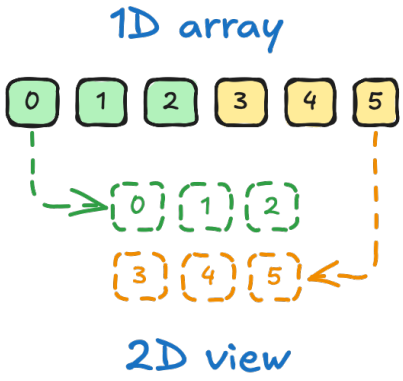
mdspan

```
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



91 NVIDIA

Once the mdspan is set up, you can access the underlying sequence using parenthesis. Just supply the row and column indices. For example, md(0, 0) accesses the very first element – 0. The passing 1, 2 into parenthesis returns the element at the second row, third column, so 5 in our case. This is a clean, intuitive way to treat a 1D buffer as a 2D array.

NVIDIA

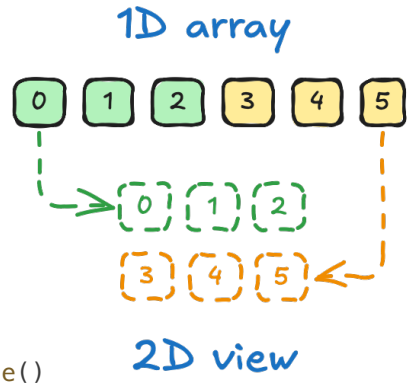
mdspan

`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()`



Besides the direct element access, mdspan also provides some handy helper functions. For instance, calling md.size() returns the total number of elements in the view - in this example, that would be 6.

mdspan

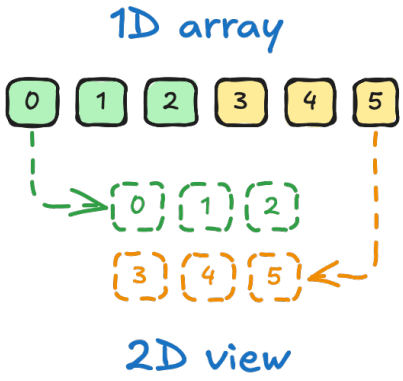
`cuda::std::mdspan::extent(r)`

- 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)`



93 NVIDIA

Additionally, mdspan lets you query the size of each dimension using the extent method. For a two-dimensional view, calling md.extent(0) gives you the height, and md.extent(1) gives you the width. In this example, md.extent(0) will return 2 and md.extent(1) will return 3. This is essentially the order at which we provided these values in the constructor.

NVIDIA

Exercise: mdspan

- 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

94 NVIDIA

Which brings us to our next exercise. This time, instead of capturing a raw pointer in our lambda, we'll capture an mdspan. That way, we can use multi-dimensional indexing directly, without manually flattening the row and column into a single index. It'll make the code cleaner and less error-prone, while maintaining the same performance.

NVIDIA

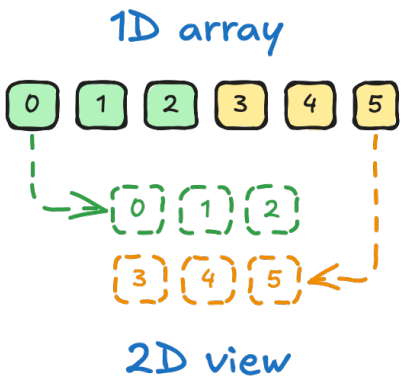
Exercise: mdspan

- 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);
    }
  });
}
```

96 NVIDIA

The key to this exercise was creating an mdspan using three parameters: 1. A raw pointer obtained from thrust::raw_pointer_cast. 2. The matrix's height. 3. The matrix's width. Because mdspan keeps track of these dimensions, our lambda no longer needs to capture height and width separately - it can just call extents function to get them. Even better, we can now write multi-dimensional indexing instead of multiplying row index by width and adding column index. It makes the code much simpler to follow.

NVIDIA

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

VIDIA

The type for cuda::std::mdspan can get pretty involved, because it lets you specify static as well as dynamic extents. That's why there exists a best practice on defining a **domain-specific type alias** instead of using mdspan type directly. For instance, we can define a type alias called temperature grid f. This alias hides the template complexity and makes your code easier to read and maintain. It also makes it clear that this mdspan is meant to handle a 2D grid of floats.

Best Practice: DRY

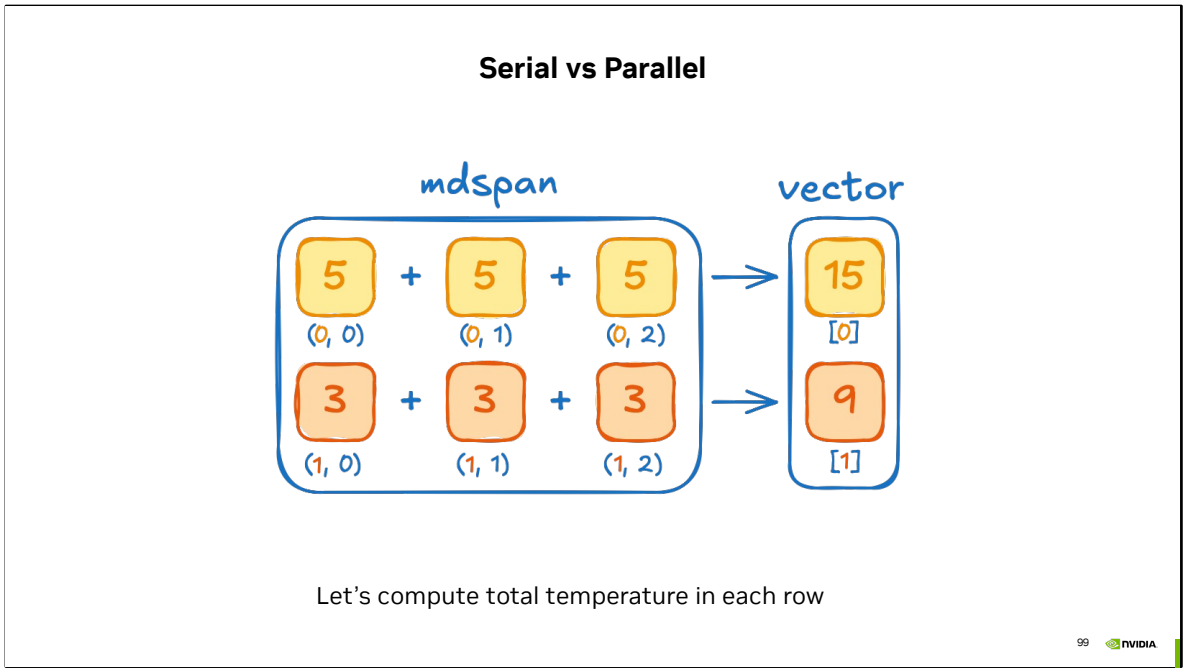
```
__host__ __device__
float compute(int cell_id, temperature_grid_f temp)
{
  int height = temp.extent(0);
  int width  = temp.extent(1);
  int column = cell_id % width;
```

- 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);
}
```

98 NVIDIA

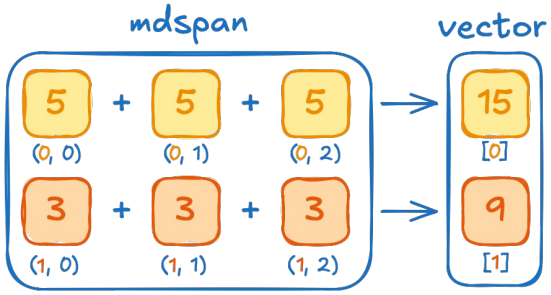
Now that we have a domain-specific alias for our mdspan, we can use it in function signatures. For example, in the code on the slide, the compute function takes a single cell index along with an mdspan. Inside, it handles all the logic and returns the next temperature. This keeps our code organized and reusable. We'll be using this pattern throughout the course.



At this point, our simulator is coming together nicely, but constantly writing out all the data to disk for visualization can be overkill. Instead, we can track something simpler, like **per-row total temperature**.
This way, we can easily see how much heat accumulates at the boundaries over time without having to dump every cell's temperature. To visualize what I mean, imagine we take each row of our 2D grid, add up the temperatures of every cell in that row, and store the sum in a 1D vector. In the example on the slide, the first row has three cells at 5 degrees each, so we store 15 in the first element of the vector. The second row has three cells at 3 degrees each, so we store 9 in the second element, and so on.
The question is, **how do we implement** this row-wise aggregation in a clean and efficient way?

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?

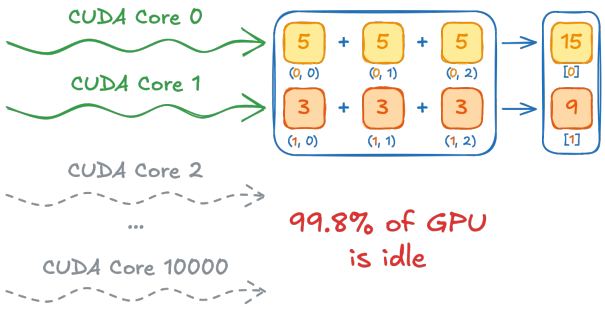
100 NVIDIA

A straightforward approach is to reach for thrust::tabulate again, but this time we tabulate over **row indices** instead of **cell indices**. Inside the lambda, we loop through each column in the row, accumulate the total temperature, and return that sum. But is this the best way to do it? Let's think about the potential performance implications.

NVIDIA

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;
  });
```

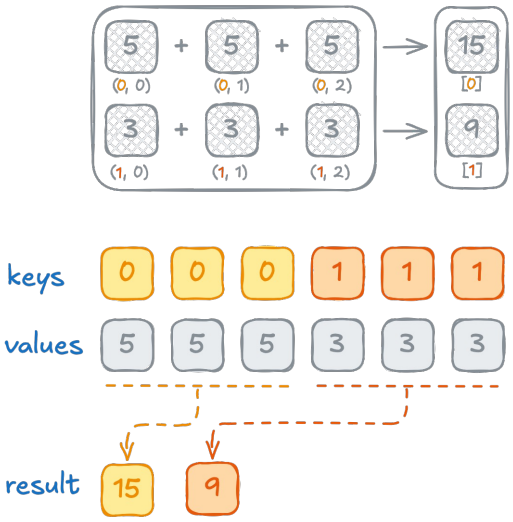


- Code on GPU does not magically become parallel
- This means that the code in lambda function is serial
- This prevents GPU from doing what it's good at – massive parallelism

101 NVIDIA

When we write our code this way, it's important to keep in mind that the GPU doesn't automatically parallelize the loop inside our lambda. Thrust only parallelizes at the level of **rows** because we're calling thrust::tabulate on row indices. If you have millions of columns but only a handful of rows, you end up with just a few threads each doing a massive amount of work. That means you're leaving most of the GPU's parallel potential on the table - effectively using less than one percent of its capacity! So, how do we fix this? We need to somehow distribute the work across many threads. In other words, we want each column in a row to be handled in parallel too. Let's explore how we can reorganize our code to achieve that.

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

102 NVIDIA

Just like before, we should begin by naming the operation we're trying to perform. If we ignore the fact that we have a row dimension, it basically looks like a standard reduction - except we're grouping values by row. That's exactly where `thrust::reduce_by_key` comes in. Unlike a plain reduce, reduce_by_key takes two input ranges: **keys** and **values**. It then groups together values that share the same key and reduces each group independently. So, in our case, the row index naturally becomes our key. All the cells in a given row share the same key, and we can sum those cell values together. This way, we spread the work across many threads while still ending up with one final sum per row.

NVIDIA

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

103 NVIDIA

To use `reduce_by_key`, we need each cell in the grid to know its row key. Straightforward way to set this up is to create a 'row IDs' array, where each entry corresponds to the row index for the cell at position `i`. In the code snippet, we do that by calling `thrust::tabulate` on our `row_ids` vector, passing a lambda that returns `i` divided by the grid width. That way, every cell in row 0 gets a key of 0, every cell in row 1 gets a key of 1, and so on.

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

104 NVIDIA

Once we have our row IDs array, we can call thrust::reduce_by_key. We pass: row_ids.begin(), row_ids.end() as the **input keys**, where each entry corresponds to the row of that cell; temp.begin() as the **input values**—these are the temperatures in our 2D grid, flattened into one dimension; thrust::make_discard_iterator() as the output keys, which we'll return in a moment, next we pass; sums.begin() as the **output values**, where the final per-row sums will be written. This way, all cells with the same row key get added together, and we end up with one sum per row in the sums vector.

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

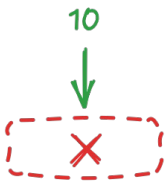
📖 2N | ✏️ 2N

105 NVIDIA

By default, `reduce_by_key` writes both keys and values into output iterators. But in our case, we already know each row corresponds to a unique key in order - so writing out those row IDs would be redundant. To save memory bandwidth, we use` thrust::make_discard_iterator()` for the key output. A discard iterator simply ignores any values written to it, so we skip storing that integer sequence and keep only the per-row sums in sums.

NVIDIA

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

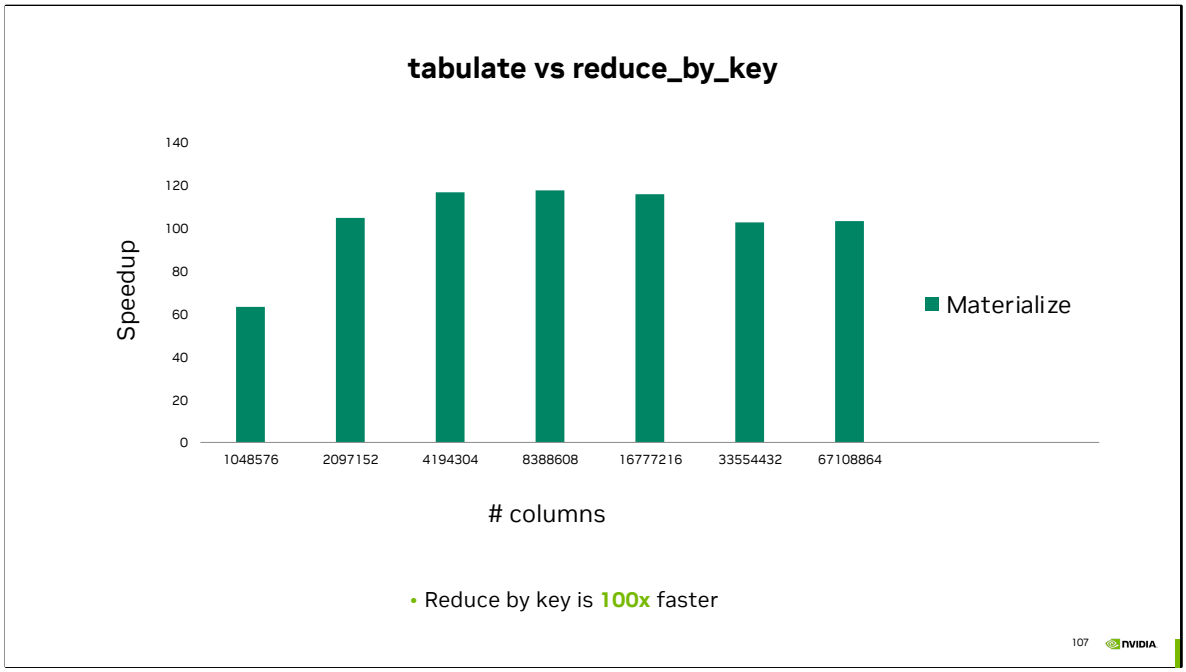
0

```
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;
}
```

106 NVIDIA

This snippet shows a simplified mental model of how a discard iterator operates. We have a wrapper struct whose assignment operator simply ignores the value you try to write. Our `discard_iterator` then returns one of these wrappers whenever you index into it, so any assignment ends up being discarded. For example, calling it[0] = 10; never stores 10 anywhere - operator=() just quietly throws the value away. That's exactly what Thrust's discard iterator does behind the scenes, letting you skip writing out data you don't actually need.

NVIDIA



These changes translate into roughly a hundredfold speedup compared to our initial approach, which relied on tabulate with a loop in the lambda. It's a striking reminder that GPUs don't magically parallelize your code - you need to design your data processing so all those GPU threads can share the workload. Whenever you see a loop inside a device lambda, ask yourself if there's a more parallel way to do it.

Exercise: Segmented Sum Optimization

- 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

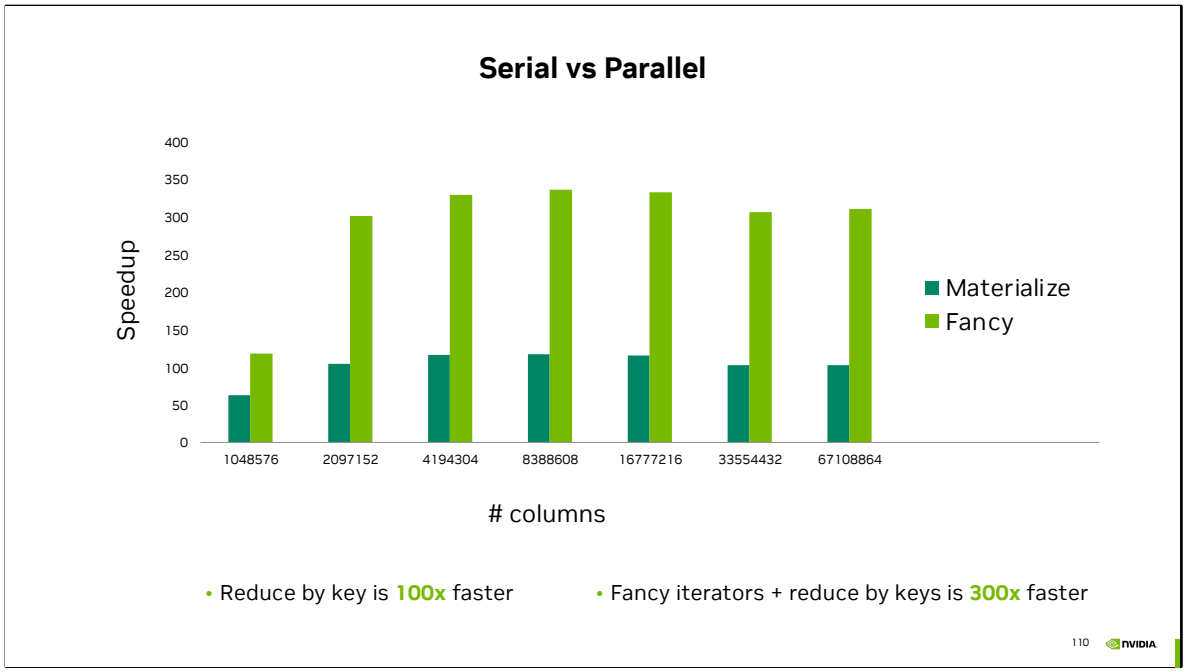
We've eliminated most sources of inefficiency, but there's still one more issue. Right now, our algorithm makes **two passes through memory -** one for the temperature values, and another for the row indices.
We don't actually need to store those row IDs in memory at all! Think about how we generated the row indices by transforming cell indices (0, 1, 2, ...) into row IDs (i / width). Instead of storing that result in a row_ids array, we can use fancy iterators. **A counting iterator** for the cells, plus a **transform iterator** that converts each cell index into a row index on the fly. That way, we never write the row IDs out to memory. Your exercise is to replace the explicit row_ids vector with these iterators, so we have only one real set of data reads and writes.

Exercise: Segmented Sum Optimization

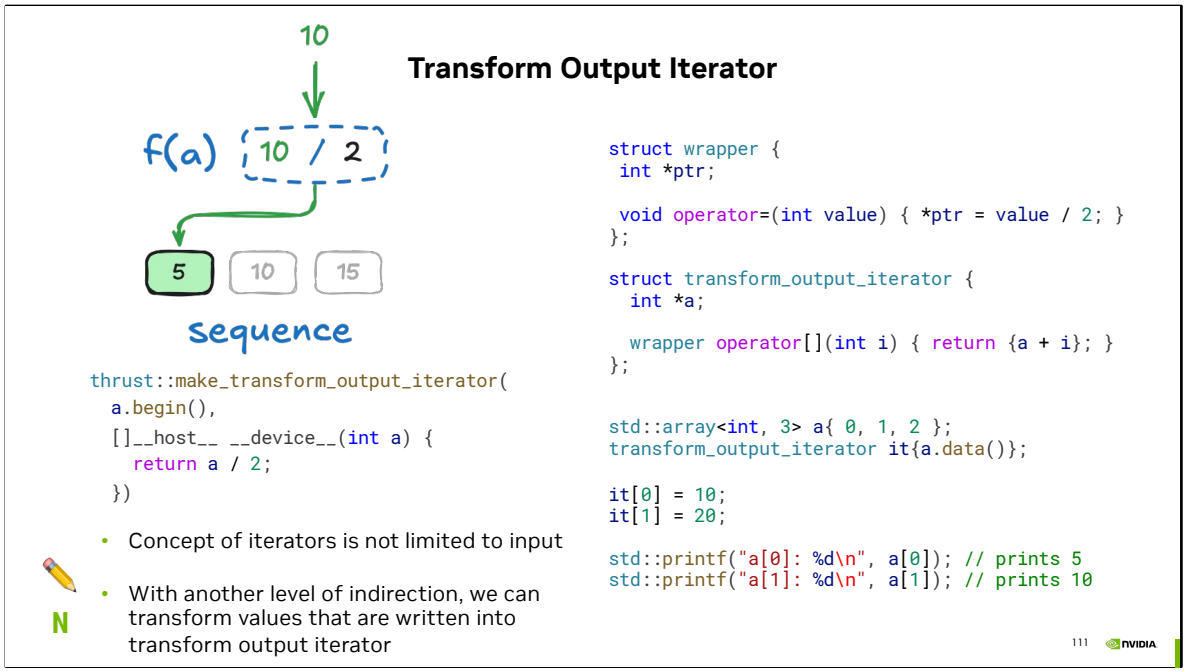
Solution

```
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());
```

The final solution uses a **transform iterator** of a **counting iterator**, so we never actually store row IDs in memory. That means reduce-by-key only reads the temperature data and writes the aggregated sums, eliminating the extra pass through memory. Practically speaking, you can't do better than that. Let's see how your change impacts performance numbers.



In this chart, we're looking at **speedups** rather than normalized elapsed times, because the difference is so large it would barely fit on the previous scale. You can see that using fancy iterators gets us around a **300x speedup** compared to our initial, more naive tabulate approach. By cutting out the extra memory reads for row indices, we're making much better use of the GPU's high bandwidth. Now that we've seen how powerful fancy iterators can be, let's explore what else we can do with them.



So far, we've been transforming values **as we read them** from an iterator. But you can also transform values **as you write** them, by adding another level of indirection on the **output** side. In the example on the slide, we return a tiny wrapper object from the iterator's subscript operator. When we assign a value to that wrapper, its assignment operator applies a transformation. In our case, it divides the incoming value by two before storing it in the underlying array. This leads to an interesting behavior. So, if you assign it at index 0 a value of 10, 10 is divided by two, and the array element actually ends up storing **5**. In many ways, a transform output iterator is **symmetrical** to the transform iterator we've used before. Except this time, the transformation happens when **writing** instead of reading. So instead of taking raw data and returning modified values, it takes the values you assign and writes a transformed version back to memory.

Exercise: Segmented Mean

- 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

112 NVIDIA

This transform output iterator idea is really handy in our row-temperature scenario. Right now, each row sum can be a very large number, which isn't always meaningful in practice - we're more interested in the *average* temperature per row. Of course, we could do another pass to divide those sums by the row width, but that would mean extra reads and writes, which we'd prefer to avoid. Instead, we can use a **transform output iterator** to convert the totals into means on the fly as we write them out. So, in this exercise, I'd like you to replace the final output iterator in the reduce-by-key call with a transform output iterator that divides each row sum by the grid width.

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);
```

113 NVIDIA

Let's take a look at the solution. We start by creating a **transform output iterator** pointing to our means vector and give it a mean_functor that divides each incoming sum by the row width. That way, when reduce-by-key writes a total into this iterator, it immediately transforms it into an average before storing. There's no extra pass over the data - just one write operation that handles both the summation and the final calculation in a single step.

Memory Spaces

Problem statement

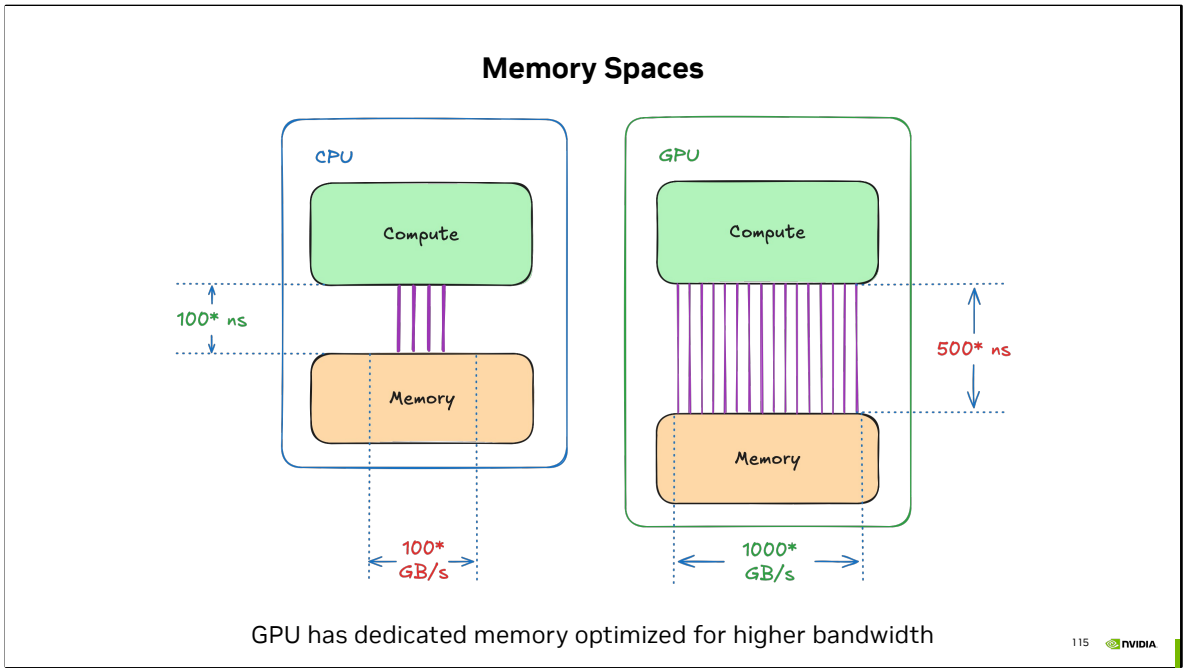
```
thrust::universal_vector<float> prev = ach::init(height, width);
thrust::universal_vector<float> next(height * width);

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

  for (int compute_step = 0; compute_step < 3; compute_step++) {
    ach::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?

114 NVIDIA

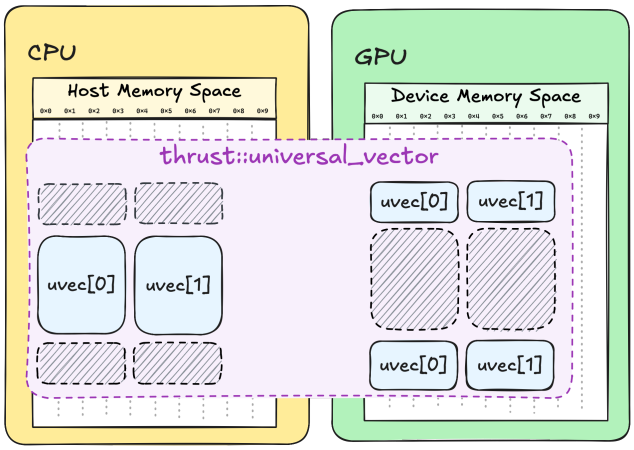
Up to this point, we've focused on optimizing our temperature simulation by cutting out unnecessary reads and writes, ultimately harnessing the GPU's massive parallelism. Now, let's take a quick look at the overall structure of our simulator. On the slide, you can see how we initialize a 'prev' buffer and a 'next' buffer, store data on disk, and then perform several simulation steps in a loop. Everything seems straightforward … except for one puzzling detail. Each time we store the data our next computations slows down - sometimes by a factor of a hundred! That's a massive performance drop, and it suggests something else is at play besides our fancy iterators or algorithm design. In fact, this hints at a deeper concern related to **memory spaces**.



Remember that slide showing the bandwidth differences between CPUs and GPUs? GPUs can handle much higher data throughput because they have their own dedicated memory, that was specifically architected for massive bandwidth. Previously, we distinguished host and device execution spaces, to know where actual code runs. To get better performance, we also have to distinguish device and host **memory spaces**, in other words, where bytes live. But that raises a question: how did our simulator run on the GPU in the first place? We haven't done anything special to allocate device memory, haven't we?

Memory Spaces

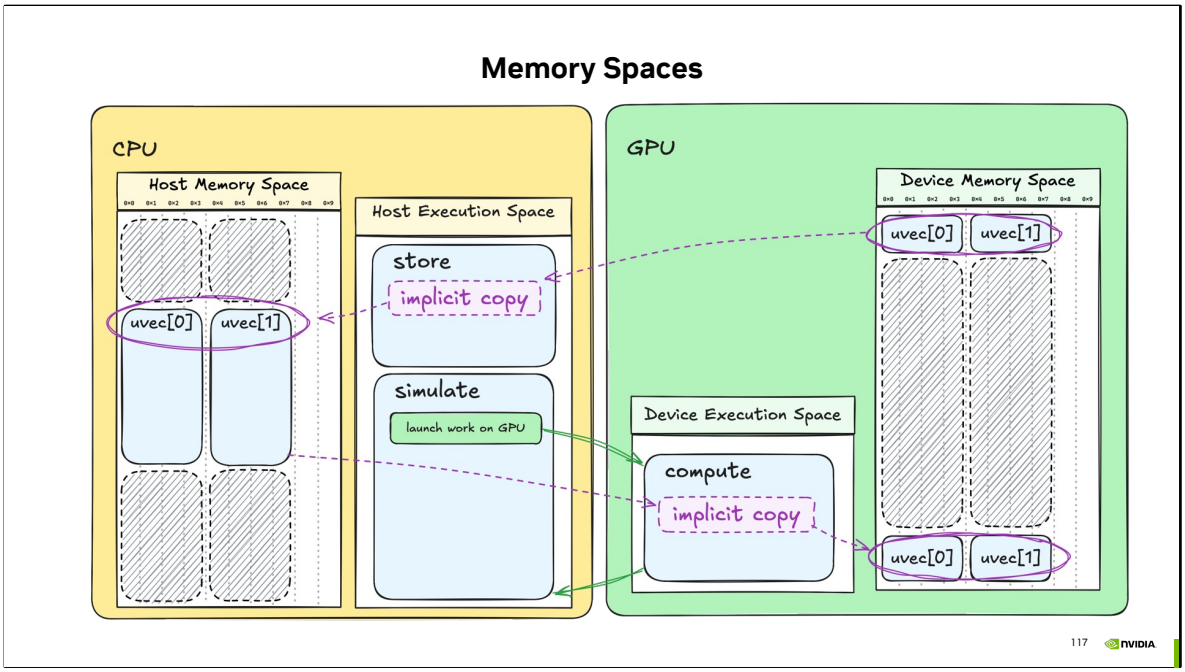
- Universal vector relies on managed memory
- Managed memory abstracts away the fact that GPU and CPU have distinct memory spaces
- Implicit memory transfer between memory spaces happens upon accessing memory



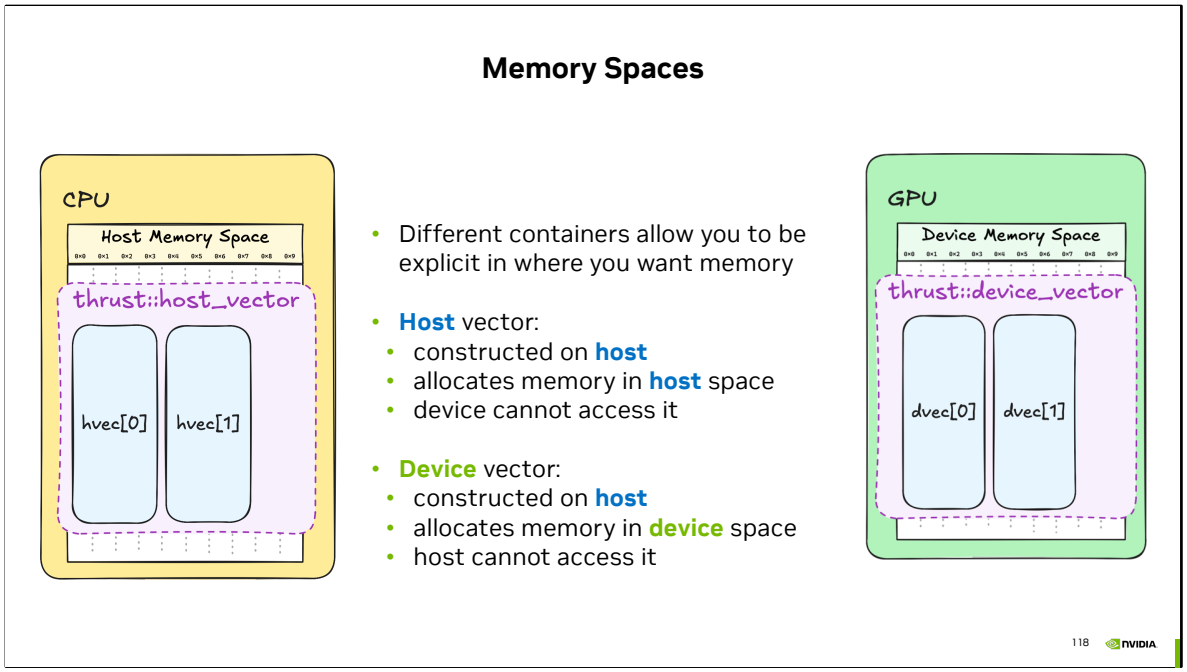
116 NVIDIA

Well, not exactly. Earlier, we switched from std::vector to thrust::universal_vector. This container uses **managed memory**, a technology that makes the CPU and GPU seem to share the same memory. Under the hood, though, the CPU and GPU still have **separate** memory spaces. Whenever the CPU or GPU accesses data, CUDA silently transfers that data to the location where it's needed. In effect, a universal vector keeps two copies of the same data: one for the CPU and one for the GPU. If the CPU reads or writes to the vector, CUDA invalidates the GPU copy, and vice versa. This saves us from writing explicit copy calls, but it also has a **performance cost**. Let's go back to our sudden slowdown and see how these hidden data transfers explain perf regression.

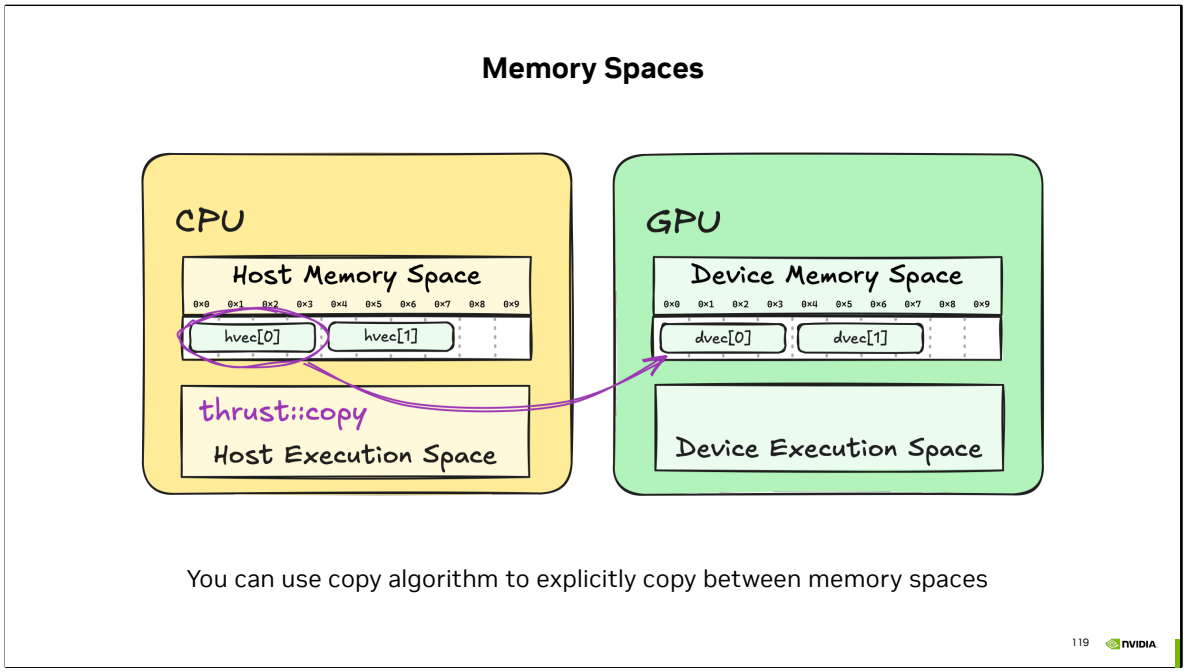
NVIDIA



Imagine the data is in the **device** memory when we call store on the **CPU**. Because the CPU needs that data, CUDA first copies it from the device to **host** memory. Once store finishes, we launch the next GPU computation step. But now the **host** copy is valid, while the GPU copy is out of date, so CUDA has to copy the data **back** to the device. That's why the first simulation step after storing takes so long - it includes the hidden cost of moving data back onto the GPU.



To solve our performance problem, we can make the GPU and CPU transfers **explicit**. Thrust offers different containers so you can decide where your data should live: **thrust::host_vector** allocates memory in host memory, so it's only accessible in the host execution space. **thrust::device_vector** allocates memory in the device memory space, so it's only accessible in the device execution space. It's important to note, that both vectors have to be constructed on the host. Anyways, by choosing the right container, we avoid the hidden back-and-forth transfers of **managed memory**. That way, we only move data across the CPU-GPU boundary when **we** decide it's necessary, which helps keep performance more predictable.



We can handle data transfers ourselves by using thrust::copy, which has the same interface as the standard library's std::copy. This way, we know exactly when data is crossing the CPU–GPU boundary, and we can control it for optimal performance.

Exercise: Use Explicit Memory Spaces

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

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


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

  for (int compute_step = 0; compute_step < 3; compute_step++) {
    ach::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

120 NVIDIA

Now we come to our final exercise in this section. This time, we'll move away from universal vector and instead use **separate** device host containers. You'll have to allocate a host vector to hold the copy of data in CPU space, and use thrust::copy to transfer that between the device and host as needed.

NVIDIA

Exercise: Use Explicit Memory Spaces

Solution

- All compute steps now take the same time to run

```
thrust::device_vector<float> d_prev = ach::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());
  ach::store(write_step, height, width, h_prev);

  for (int compute_step = 0; compute_step < 3; compute_step++)
  {
    ach::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
...
```

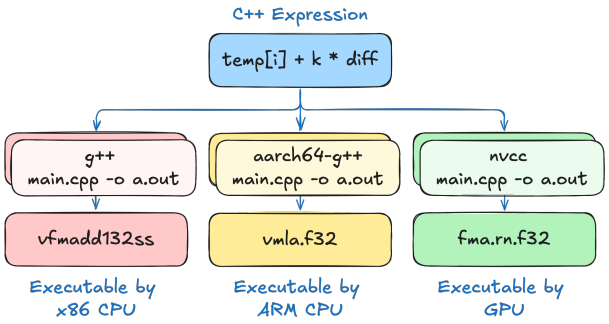
121 NVIDIA

Let's walk through the solution step by step: we replaced universal vector with separate **device** and **host** vectors. The device vectors (d_prev and d_next) re allocated in GPU memory, while host vector h_prev lives in CPU memory. The `d_`, and `h_` prefix is quite common to denote the difference in memory spaces. Then, each time we need to store results to disk we first **explicitly copy** from d_prev to h_prev using thrust::copy. We then pass h_prev to the store function, which records the data on the CPU side. Now, there's no hidden transfer since both d_prev and d_next live entirely on the GPU. With this change, data only crosses the CPU–GPU boundary at times we explicitly choose, eliminating the surprising 100x slowdown.

NVIDIA

Takeaways

Use NVCC compiler



122 NVIDIA

Before proceeding to the next section, here's a quick recap of the main takeaways. Use NVCC to compile your code for NVIDIA GPUs.

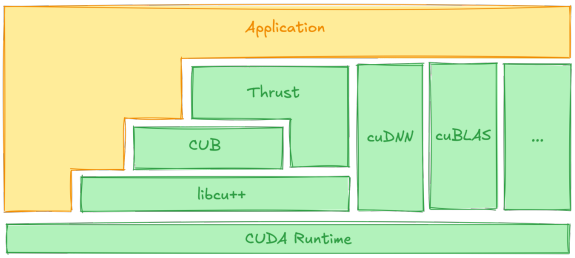
NVIDIA

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.



123 NVIDIA

Leverage **accelerated libraries** whenever you can. Existing libraries are already tuned to make the most of GPU hardware. For example, use **Thrust** for general-purpose parallel algorithms and container management. Use **cuSPARSE** when you need GPU-accelerated sparse linear algebra functions. Use **MatX** for array-based numerical computing, etc.

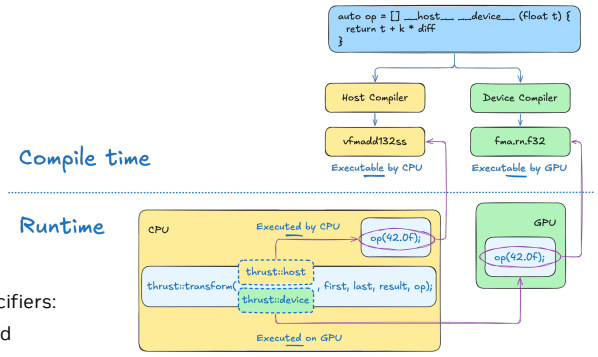
NVIDIA

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

124 NVIDIA

When writing your CUDA code, remember the difference between execution specifiers and policies: 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.

NVIDIA

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.

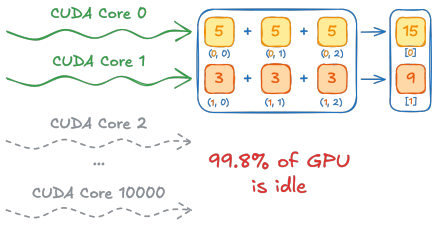
Use `__host__ __device__` execution space specifiers:

- 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



125 NVIDIA

Avoid serialization as much as possible.

NVIDIA

Takeaways

Use NVCC compiler

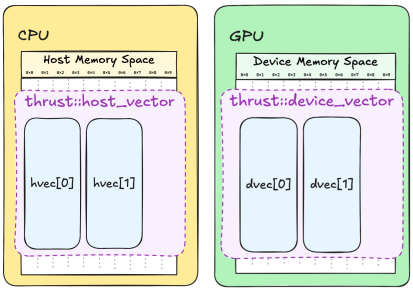
Use accelerated libraries as much as possible:
- Thrust for general-purpose parallel algorithms
- cuSPARSE for linear algebra
- MatX for array-based numerical computing
- etc.

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



Where possible, use **explicit memory spaces** to avoid unexpected slowdowns. Rather than relying on implicit managed memory transfers, define separate **host** and **device** vectors and manage data movement yourself. This way, you're always in control of when and how data travels between the CPU and GPU.

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.

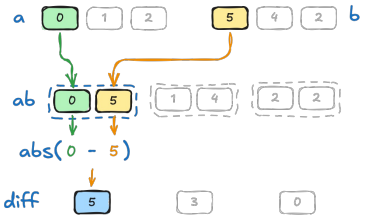
Use `__host__ __device__` execution space specifiers:
- to specify where given function **could** be executed

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



127 NVIDIA

And lastly, fancy iterators give you a powerful way to **extend existing parallel algorithms** to your unique use cases. By encapsulating custom logic, you can reuse Thrust's highly optimized building blocks without having to rewrite entire algorithms from scratch.

NVIDIA