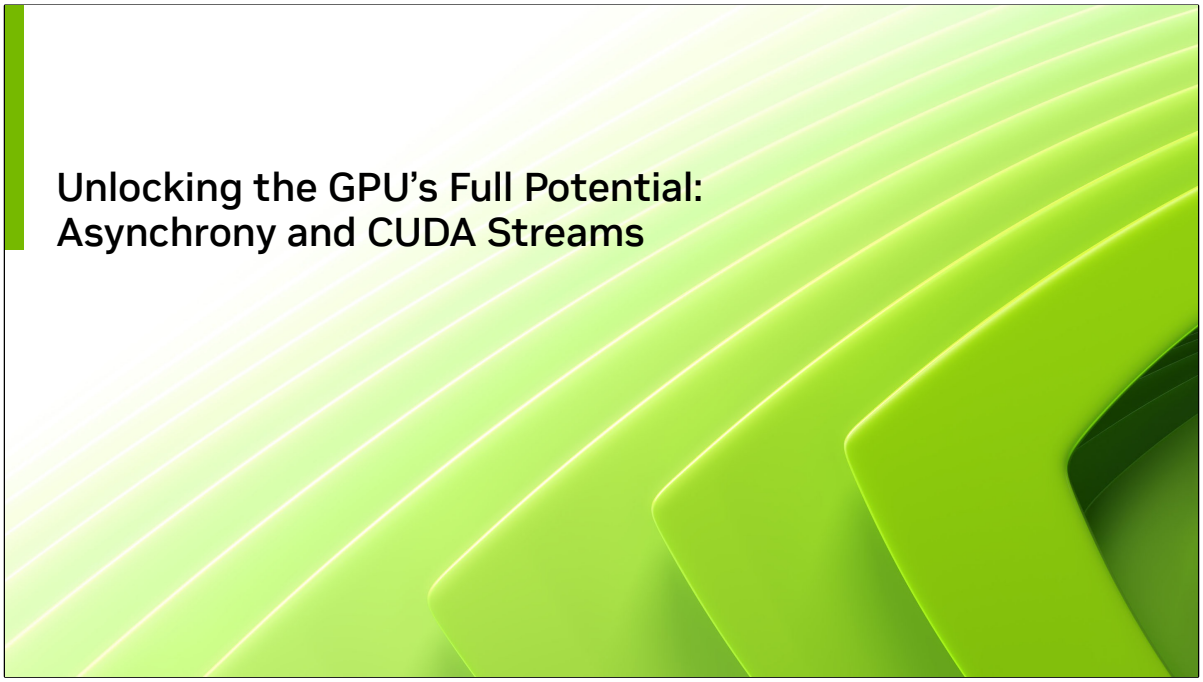
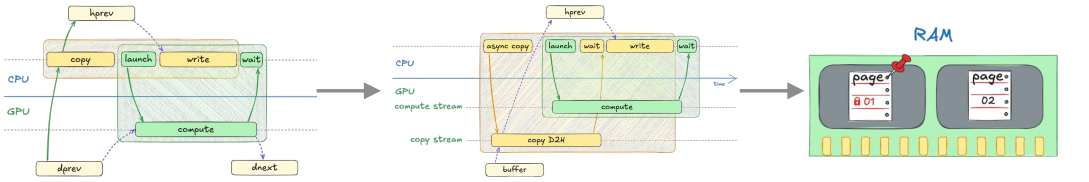


In the previous section, we covered essential concepts like execution and memory spaces, vocabulary data types, and parallel algorithms. We also created a functional 2D heat equation simulator. Although it currently runs on the GPU, our simulator still isn't making full use of the system's capabilities. In this section, we'll explore a few additional techniques to optimize its performance.

Section at a Glance



Learn the notion of 2.2 **Asynchrony** as we overlap computation and IO to achieve 2x speedup

Learn the notion of 2.3 **CUDA Streams** as we overlap computation with memory transfers

Learn the notion of 2.4 **Pinned Memory** as we make memory transfers asynchronous

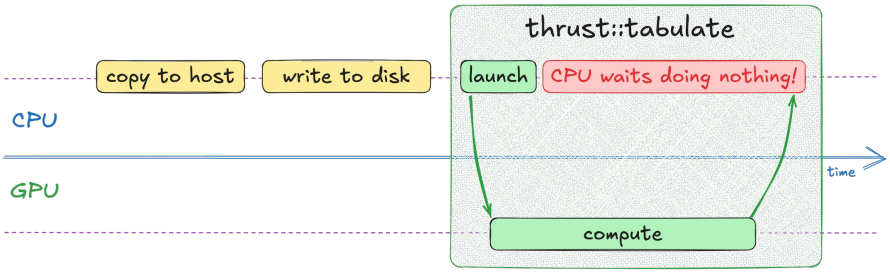
2 NVIDIA

Here's how this section is organized. First, we'll explore the concept of asynchrony, which lets us overlap computation with I/O. Next, we'll learn how CUDA streams enable concurrent CPU-GPU data transfers. Finally, we'll explore new aspects of the memory subsystem that support efficient data movement.

NVIDIA

Recap of Current State

- Copy previous data to CPU and write it to disk
- Invoke **thrust::tabulate** that reads previous state and writes next one



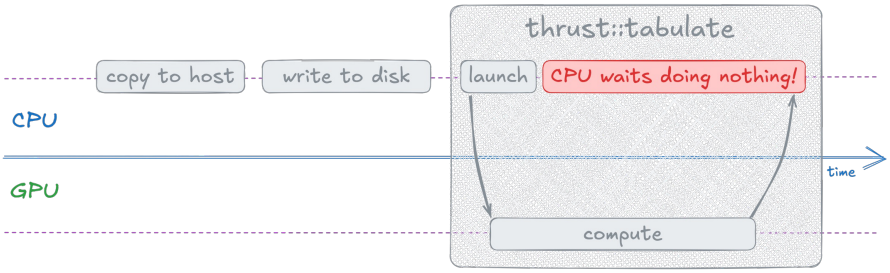
3 NVIDIA

Let's review the current state of our heat equation simulator. First, we copy the data from the GPU to the host, then we write it to disk, and finally, we use thrust::tabulate to compute the next set of temperatures on the GPU.

NVIDIA

Idle Resources

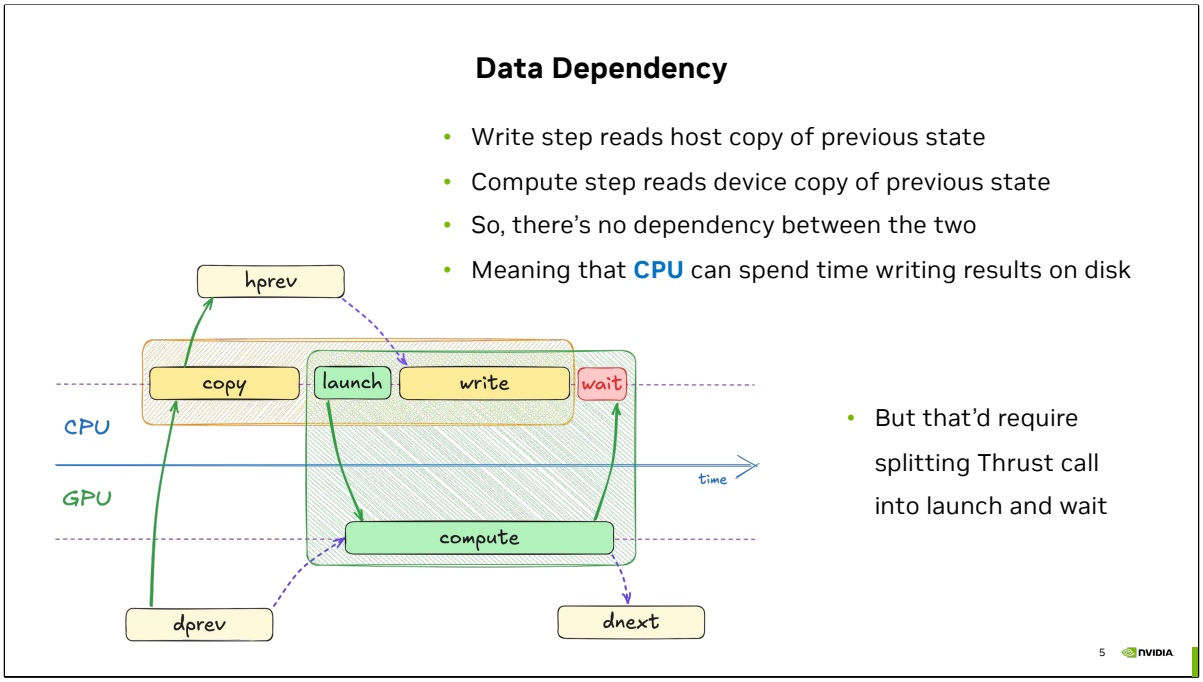
- Under the hood, Thrust launches work on GPU and waits for it to finish (synchronous)
- Writing efficient heterogeneous program means utilizing all recourses, including **CPU**
- Can we find anything useful for **CPU** to do while **GPU** is busy?



4 NVIDIA

Under the hood, Thrust launches work on the GPU and waits for it to finish. Because thrust::tabulate doesn't return control until the transformation is complete, the CPU stays idle. This is a missed opportunity! Writing an efficient heterogeneous program means making full use of all resources. This includes the CPU as well. So, can we find anything meaningful for the CPU to do while the GPU is busy?

NVIDIA



To answer this question, let's take a look at the data dependencies. The write step reads from the host copy of the previous temperatures, while the compute step reads from the device copy. Because these operations use separate data, they don't depend on each other. As a result, the CPU can write results to disk while the GPU computes the next temperatures. To achieve this overlap, we'd ideally split the Thrust call into separate launching and waiting steps. Unfortunately, Thrust doesn't support that. The good news is that Thrust doesn't implement its algorithms directly. Instead, it relies on another core library called CUB to handle work on the GPU. Let's take a closer look at CUB to see if it can help us overlap compute and IO.

CUB

Sorting

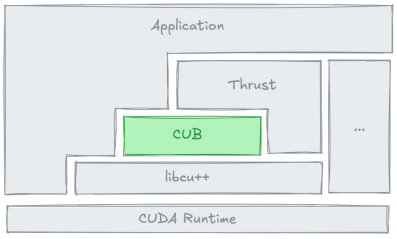
- `cub::DeviceTransform`
- `cub::DeviceRadixSort`
- `...`

Iterators

- `cub::CacheModifiedInputIterator`
- `cub::CacheModifiedOutputIterator`
- `...`

Types

- `cub::FutureValue`
- `cub::DoubleBuffer`
- `...`



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

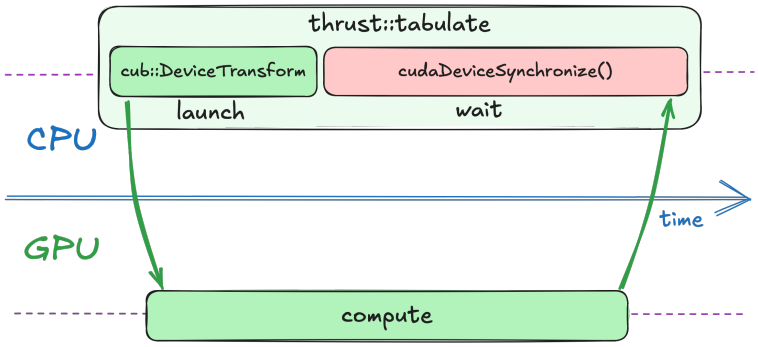
6 NVIDIA

CUB offers many of the same parallel algorithms found in Thrust, but it doesn't follow the standard C++ API. It's also CUDA-specific, meaning it doesn't provide any host-side implementations like Thrust does.

NVIDIA

Synchronous vs Asynchronous

- Thrust calls into another layer of CUDA Core Libraries called CUB
- Unlike Thrust, CUB is asynchronous (control flow doesn't wait for algorithm to finish)
- To synchronize with all asynchronous work on GPU, Thrust uses `cudaDeviceSynchronize`



7 NVIDIA

Under the hood, Thrust calls into another layer of the CUDA Core Libraries called CUB. CUB is asynchronous, so once you launch an operation, it immediately hands control back to the CPU instead of waiting for the GPU to finish. After launching CUB algorithm, Thrust uses cudaDeviceSynchronize to wait for CUB algorithm completion. This cudaDeviceSynchronize is part of the CUDA Runtime. It essentially blocks the CPU thread until every preceding GPU operations have completed.

NVIDIA

CUB is Asynchronous

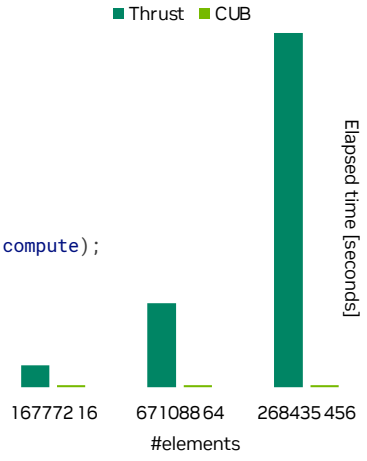
Thrust

```
auto begin = std::chrono::high_resolution_clock::now();
thrust::tabulate(thrust::device, out.begin(), out.end(), compute);
auto end = std::chrono::high_resolution_clock::now();
```

CUB

```
auto begin = std::chrono::high_resolution_clock::now();
auto cell_ids = thrust::make_counting_iterator(0);
cub::DeviceTransform::Transform(cell_ids, out.begin(), num_cells, compute);

auto end = std::chrono::high_resolution_clock::now();
```

- CPU doesn't wait for the transformation to finish before executing the next instruction (recording end time)
- That's why CUB elapsed time doesn't scale with problem size



8 NVIDIA

Asynchrony is a key concept in GPU programming, and the example on this slide shows why. In the first snippet, we use thrust::tabulate. Thrust is synchronous, so our timer doesn't stop until the GPU has completely finished the work. In the second snippet, we use CUB's DeviceTransform function, which is asynchronous. That means it returns control to the CPU right away, even if the GPU is still computing. As a result, when we record the end time, the GPU hasn't actually finished yet, so it looks like CUB takes no time at all. Increasing the problem size doesn't change that timing, because the CPU never waits for the GPU to finish. This illustrates what asynchronous execution really means: the CPU and GPU can work in parallel. That's why timing measurements can be misleading if you don't explicitly synchronize before checking how long the GPU work took.

NVIDIA

Explicit Device Synchronization

Thrust

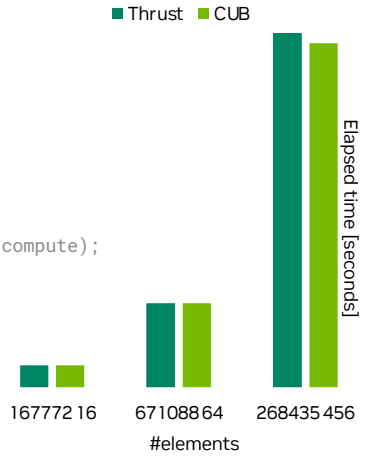
```
auto begin = std::chrono::high_resolution_clock::now();
thrust::tabulate(thrust::device, out.begin(), out.end(), compute);
auto end = std::chrono::high_resolution_clock::now();
```

CUB

```
auto begin = std::chrono::high_resolution_clock::now();
auto cell_ids = thrust::make_counting_iterator(0);
cub::DeviceTransform::Transform(cell_ids, out.begin(), num_cells, compute);
cudaDeviceSynchronize();
auto end = std::chrono::high_resolution_clock::now();
```

`cudaDeviceSynchronize`

- is part of CUDA Runtime
- blocks until all work on GPU finishes
- Makes CUB version synchronous as well

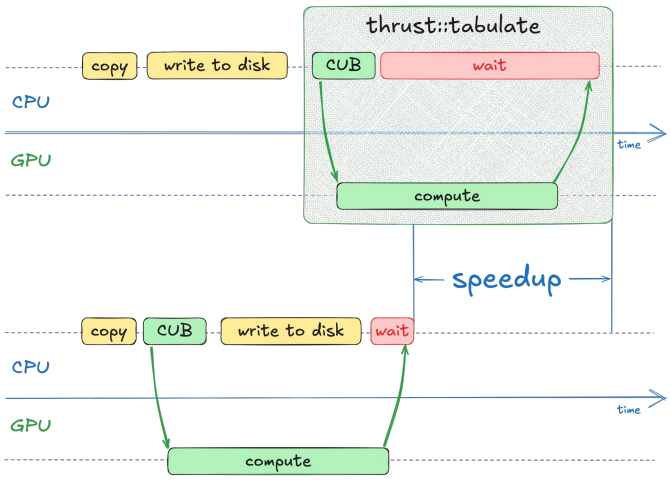


9 NVIDIA

So, let's do exactly that. By adding the cudaDeviceSynchronize call we mentioned earlier, we force the CPU to wait until all GPU work is complete. Once we add this synchronization, both CUB and Thrust show equivalent performance results. The question then becomes: how do we take advantage of this asynchronous behavior?

NVIDIA

Synchronous vs Asynchronous



- By using asynchronous CUB API along with CUDA Runtime directly, we can keep CPU busy with writing results on disk
- This should result in speedup

10 NVIDIA

Now we can finally split our Thrust call into separate launch and wait steps. When we launch asynchronous work with CUB, the CPU is freed immediately, allowing us to start writing data to disk while the GPU continues computing.
We then synchronize with the GPU only after it finishes. Since we would have waited for the GPU anyway, we're not losing any time. Instead, the wait step becomes shorter, leading to more efficient use of the system. As a result, we can expect a significant performance boost from this approach."

NVIDIA

Exercise: Compute-IO Overlap

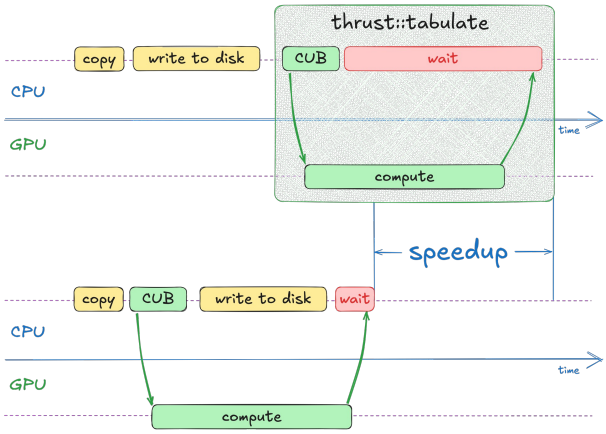
1. Replace `thrust::tabulate`

```
thrust::tabulate(
  thrust::device, out.begin(), out.end(),
  compute);
```

with `cub::DeviceTransform::Transform`

```
auto cell_ids =
  thrust::make_counting_iterator(0);
cub::DeviceTransform::Transform(
  cell_ids, out.begin(), num_cells,
  compute);
```

2. Add `cudaDeviceSynchronize` after storing results on disk to match scheme on the right



02.02-Asynchrony/02.02.02-Exercise-Compute-IO-Overlap.ipynb

11 NVIDIA

Which brings us to our next exercise. This time, you'll replace thrust::tabulate with cub::DeviceTransform to make the computation asynchronous. You'll also add a cudaDeviceSynchronize call after writing the results to disk, following the scheme shown on the right.

NVIDIA

Exercise: Compute-IO Overlap

Solution

- CUB returns control immediately without waiting for transformation to finish

```
thrust::copy(dprev.begin(), dprev.end(), hprev.begin());

for (int step = 0; step < steps; step++)
{
  simulate(height, width, dprev, dnext); - - ▶ void simulate(int width,
  dprev.swap(dnext);                                    int height,
}                                                       const thrust::device_vector<float> &in,
                                                              thrust::device_vector<float> &out)
                                                {
store(write, height, width, hprev);               // ...
cudaDeviceSynchronize();                          auto cell_ids = thrust::make_counting_iterator(0);
                                                  cub::DeviceTransform::Transform(
                                                    cell_ids, out.begin(),
                                                    width * height,
                                                    compute);
                                                }
```


Let’s review the solution. We replace thrust::tabulate with cub::DeviceTransform in the simulate function, since tabulate is essentially just a transformation of a counting iterator. Once the GPU computation is launched asynchronously, we can do other CPU work - in this case, writing data to disk.

Exercise: Compute-IO Overlap

Solution

- CUB returns control immediately without waiting for transformation to finish
- So, we must add `cudaDeviceSynchronize` explicitly

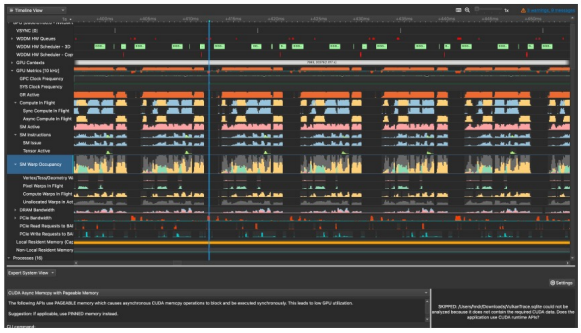
```
thrust::copy(dprev.begin(), dprev.end(), hprev.begin());

for (int step = 0; step < steps; step++)
{
  simulate(height, width, dprev, dnext);
  dprev.swap(dnext);
}

store(write, height, width, hprev);
cudaDeviceSynchronize();
```

```
void simulate(int width,
              int height,
              const thrust::device_vector<float> &in,
                    thrust::device_vector<float> &out)
{
  // ...
  auto cell_ids = thrust::make_counting_iterator(0);
  cub::DeviceTransform::Transform(
    cell_ids, out.begin(),
    width * height,
    compute);
}
```

13 NVIDIA

After writing results on disk, we add cudaDeviceSynchronize to wait for GPU to finish all work.

Nsight Systems



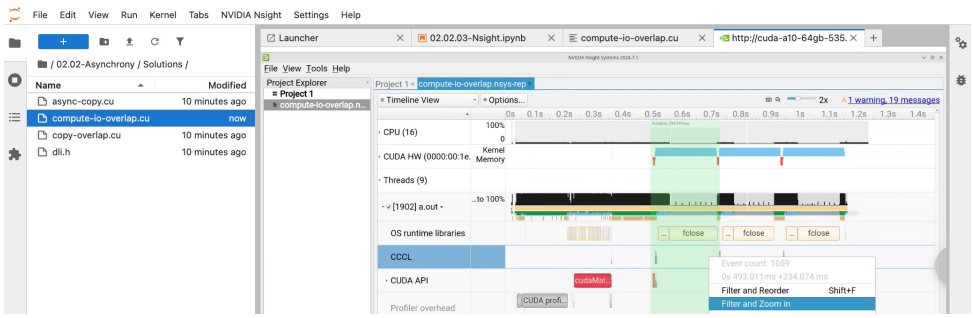
- Nsight Systems provides system-wide view of **GPU** and **CPU** activities
- It visually represents asynchronous compute and memory transfers
- It allows us to visually identify optimization opportunities

As we dive deeper into asynchronous execution, profiling and debugging can get tricky. It's not always clear when tasks overlap, or if the CPU and GPU are being fully utilized. This is where Nsight Systems steps in. Nsight Systems gives you a system-wide view of both GPU and CPU activities, visually representing asynchronous operations. By showing you exactly when and where each task runs, it highlights potential bottlenecks and uncovers new opportunities for optimization. Let's take a closer look at how we can use this tool to fine-tune our asynchronous workflows.

Exercise: Profile Your Code with Nsight Systems

Look around the timeline of *Desktop/nsight-reports/compute-io-overlap.nsys-rep* to identify:

- when GPU compute is launched
- when CPU writes data on disk
- when CPU waits for GPU
- when data is transferred

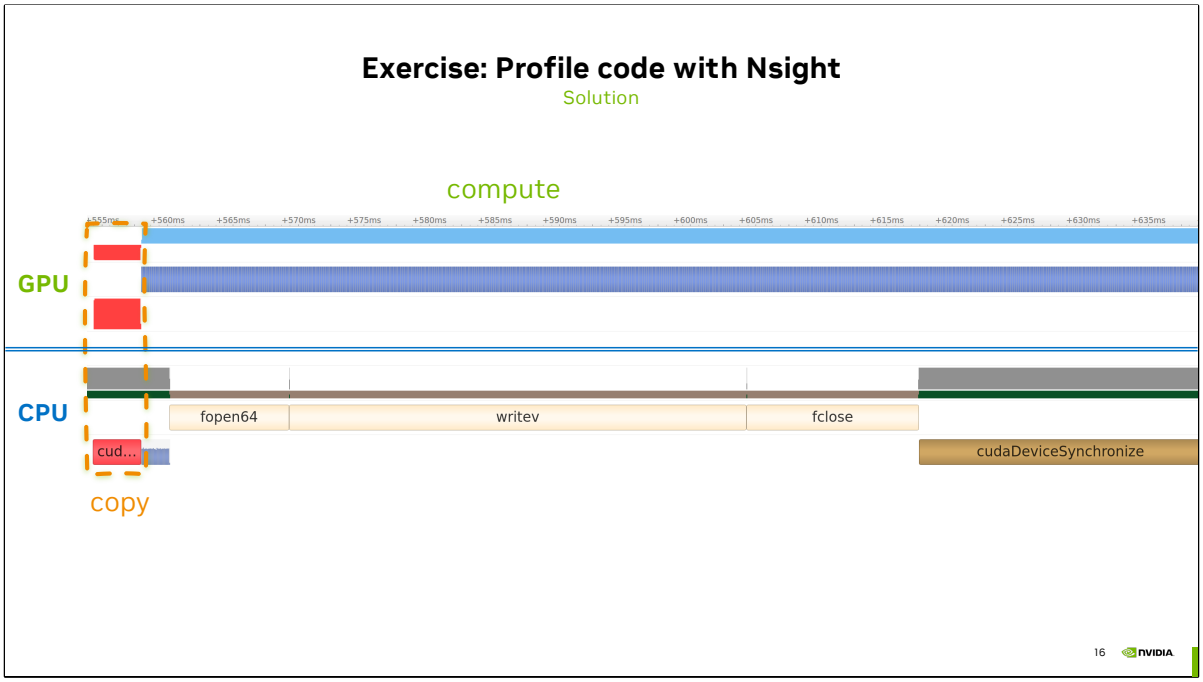


02.02-Asynchrony/02.02.03-Exercise-Nsight.ipynb

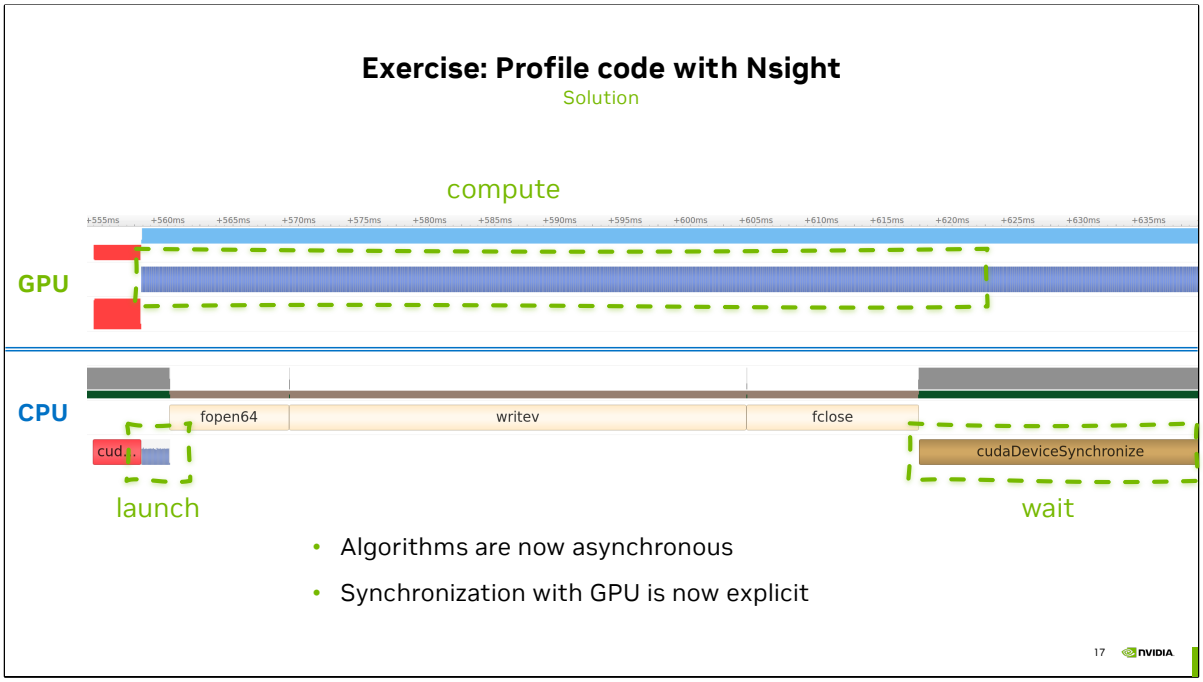
15 NVIDIA

Overall, your task is to look around the timeline of the profile that we just generated and identify: when GPU compute is launched; when CPU writes data on disk; when CPU waits for GPU; and finally, when the data is transferred between CPU and GPU.

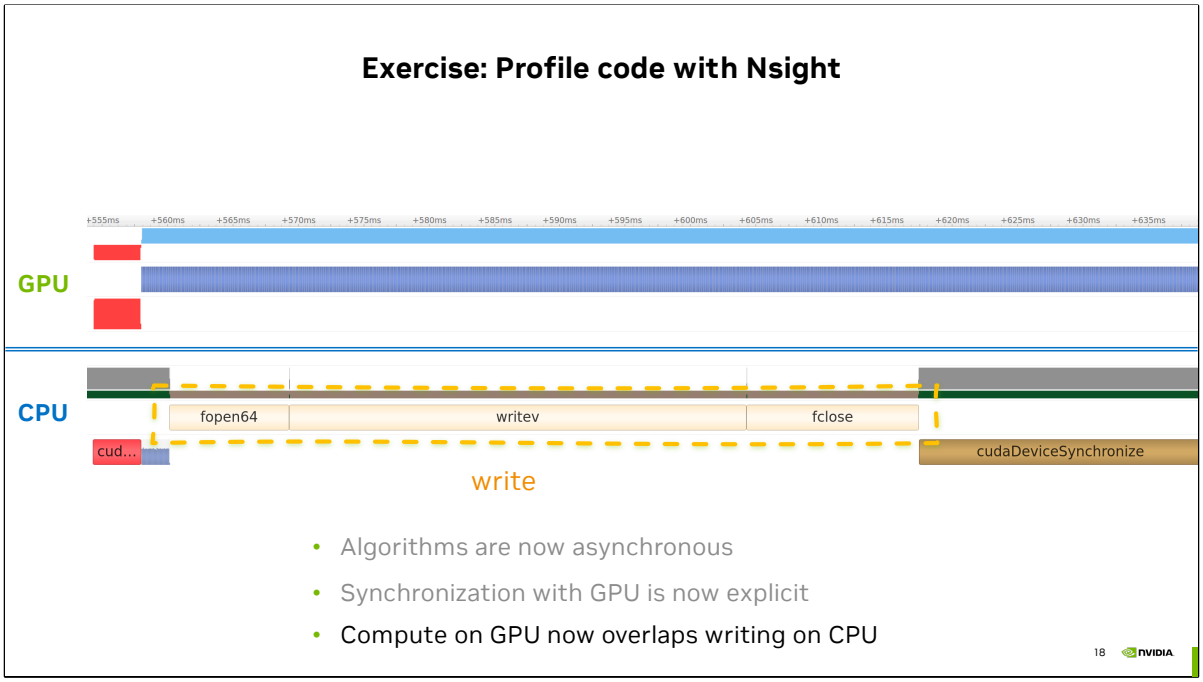
NVIDIA



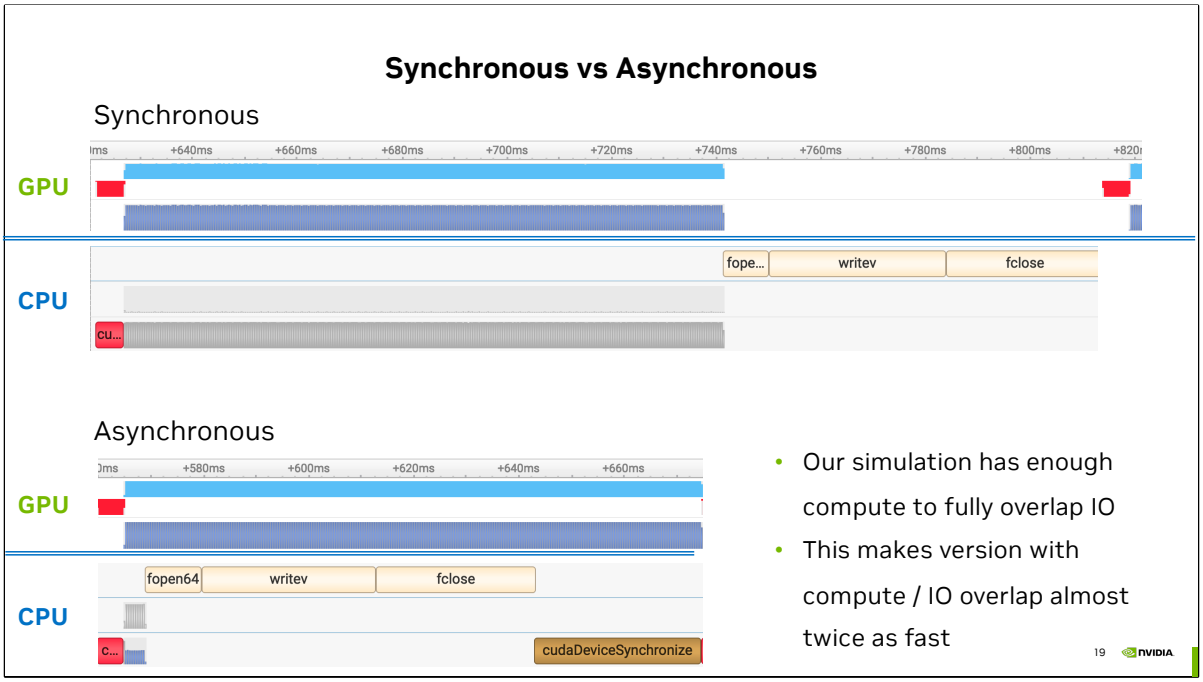
Let’s take a look at the solution. Our write step starts with a copy from GPU to CPU memory space.



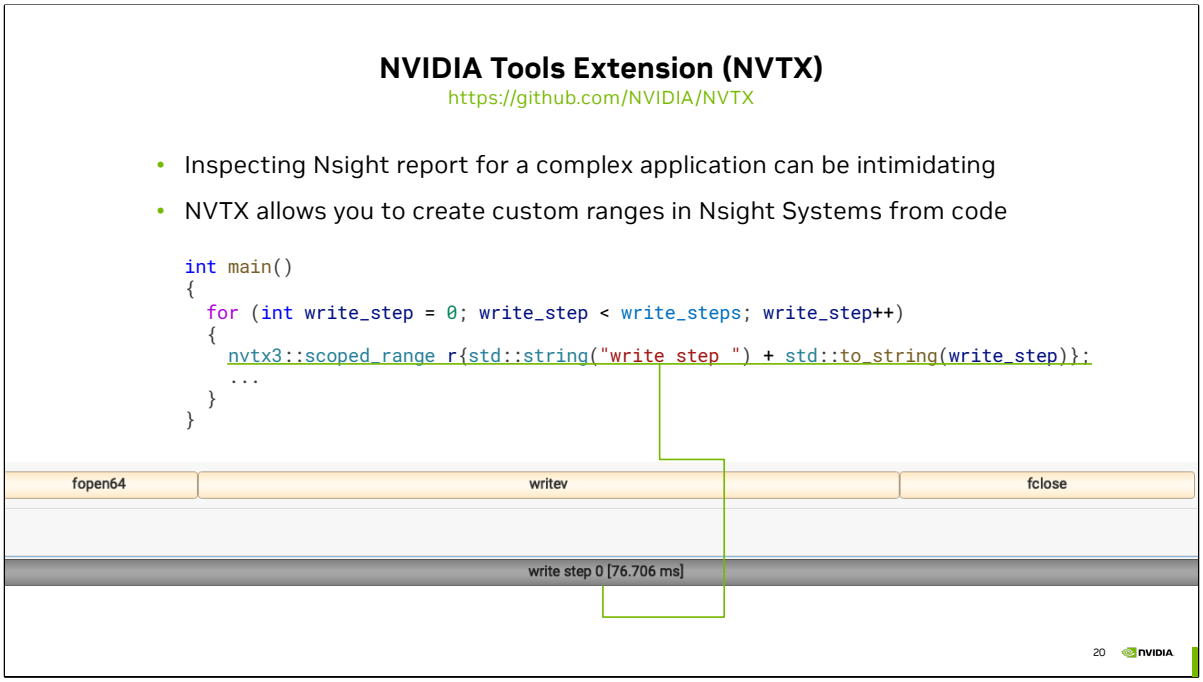
Additionally, notice how the compute is initiated by the CPU but actually runs on the GPU. The cudaDeviceSynchronize call, which marks the end of each iteration, is clearly visible as well. You can see how it holds up the CPU until the final kernel has finished executing on the GPU, before handing control flow back to the CPU.



If you're familiar with operating system calls, you'll spot fopen, writev, and fclose on the CPU timeline. These indicate that the CPU was writing results to disk. Notice how this disk I/O overlaps with GPU computation, which you can see happening at the same time. Thanks to profiling with Nsight Systems, we don't have to guess if overlapping actually occurs. We can just see it right here in the timeline.



If you profile the earlier version of our simulator, you'll see a clear performance difference. The synchronous API blocks after each call, which delays disk writes until all computations are finished, so no overlap. In contrast, the new asynchronous version fully overlaps computation and I/O, making the disk writes effectively 'free' and yielding about a 2x speedup. That said, analyzing these differences still involves matching what you see in the timeline with your code, which can be cumbersome. Fortunately, there's a way to simplify profiling even further.

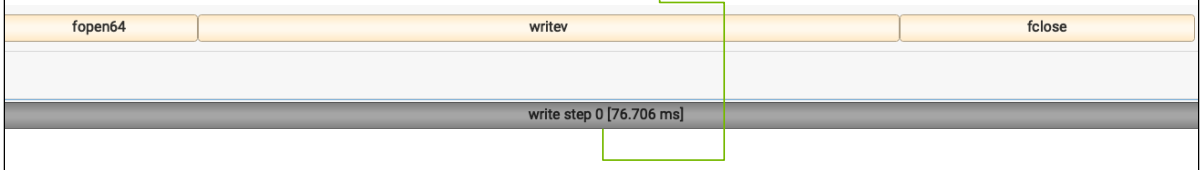


When you're working on a complex application, interpreting Nsight report can be overwhelming. That's where NVTX, the NVIDIA Tools Extension, comes in. NVTX allows you to insert custom markers and ranges directly into your code, so that when you view the Nsight report, your labels appear in the timeline. In the code on the slide, we use `nvtx3::scoped_range` to label each write step. The range starts at the beginning of the loop iteration and ends automatically when it goes out of scope. This helps you instantly map parts of the timeline to the code you write.

Exercise: Use NVTX

- Use `nvtx3::scoped_range` to annotate copy, compute, write, and wait steps
- Profile your application with Nsight Systems and locate specified ranges

```
int main()
{
  for (int write_step = 0; write_step < write_steps; write_step++)
  {
    nvtx3::scoped_range r{std::string("write step ") + std::to_string(write_step)};
    ...
  }
}
```



02.02-Asynchrony/02.02.04-Exercise-NVTX.ipynb

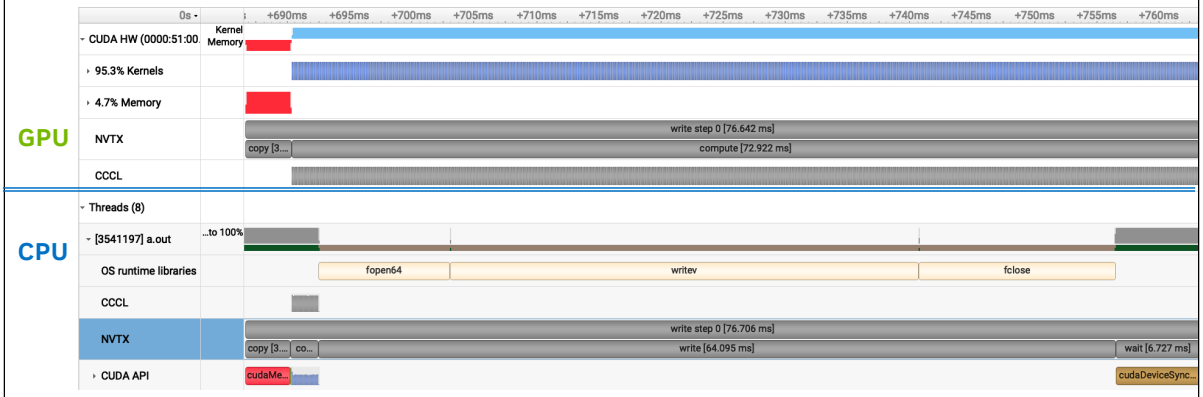
21 NVIDIA

Which brings us the the next exercise. This time, you'll have to annotate copy, compute, write, and wait steps using NVTX ranges. Then, profile the simulator with Nsight systems and try locating specified ranges.

NVIDIA

Exercise: Use NVTX

Solution

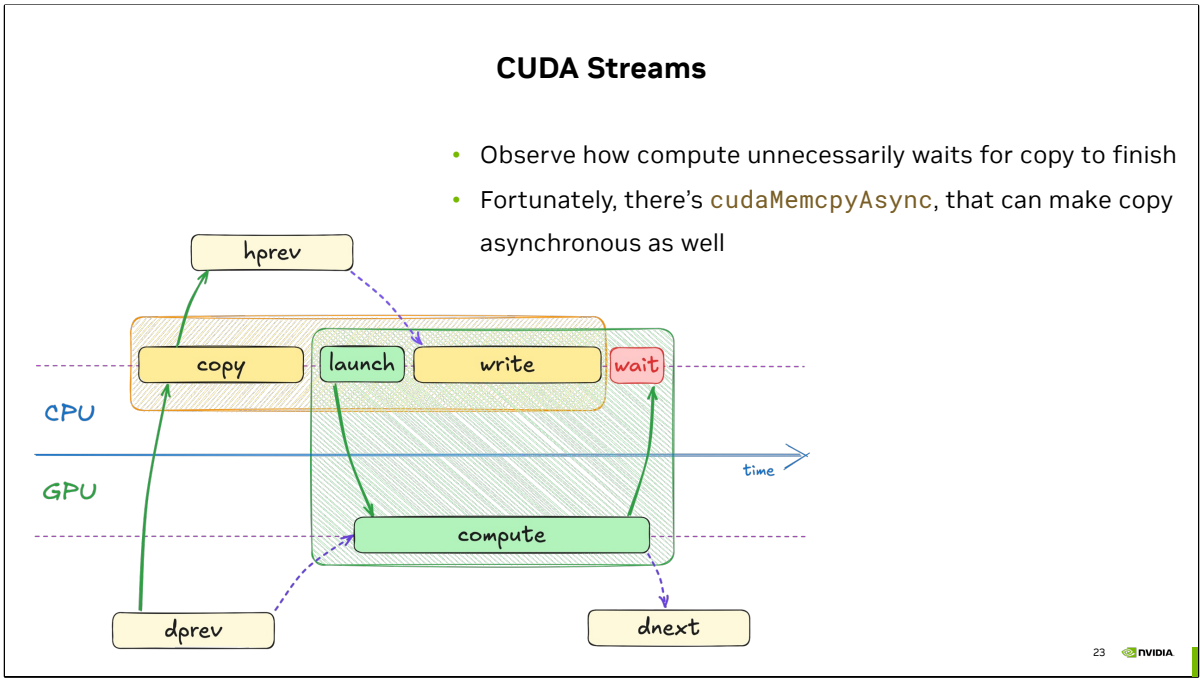


- Now all the steps are much easier to locate

22 NVIDIA

After making those changes, your Nsight timeline should look something like this. Notice how each labeled range clearly marks where every step begins and ends. This visual labeling makes it much easier to spot precisely what's happening at each point in your application

NVIDIA



At this point, we've learned how to use asynchrony to overlap CPU and GPU tasks, leading to a big performance boost. Our code now copies data from the GPU to the CPU, launches an asynchronous computation with CUB, and then writes results to disk. However, there's still room for improvement: the GPU compute doesn't actually have to wait for the copy to finish. We can apply the same overlapping strategy to the copy and compute steps. But this will require the copy to be asynchronous. Fortunately, Thrust itself doesn't have any special mechanism to transfer data. Under the hood, it relies on the CUDA Runtime function called cudaMemcpyAsync.This function can copy data between host and device memory without blocking. Let's take a closer look at how we can use it to overlap our copy and compute steps.

CUDA Streams

- `cudaMemcpyAsync` is the magic behind `thrust::copy`
- Warning:
  - Not type-save
  - Operates in bytes not elements
  - Explicitly specified copy direction
  - Asynchronous errors are easy to miss

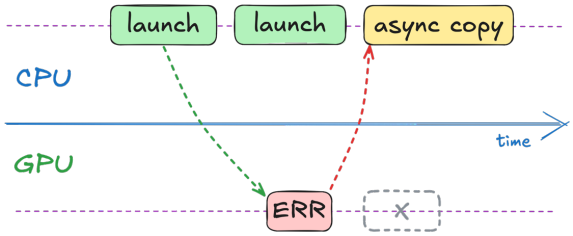
```
cudaError_t cudaMemcpyAsync(
  void*          dst,   → Destination memory address
  const void*    src,   → Source memory address
  size_t         count, → Size in bytes to copy
  cudaMemcpyKind kind   → cudaMemcpyHostToHost
                        → cudaMemcpyHostToDevice
                        → cudaMemcpyDeviceToHost
                        → cudaMemcpyDeviceToDevice
);
```

24 NVIDIA

Let's begin with the cudaMemcpyAsync interface, which essentially powers thrust::copy. It handles asynchronous data transfers between host and device memory, but there are a few important pitfalls to keep in mind. First, cudaMemcpyAsync isn't type-safe - it won't verify that the source and destination types match. Second, it operates in bytes rather than elements, so you need to multiply the number of elements by the size of each element to avoid mistakes. Third, you must explicitly indicate whether the transfer is host-to-device, device-to-host, or device-to-device using the cudaMemcpyKind parameter. Finally, cudaMemcpyAsync returns an error code, which you should always check. In a moment, we'll take a closer look at what these errors might mean and how to handle them.

CUDA Errors

- What kind of error does `cudaMemcpyAsync` return?
- The error can be caused any prior asynchronous operation



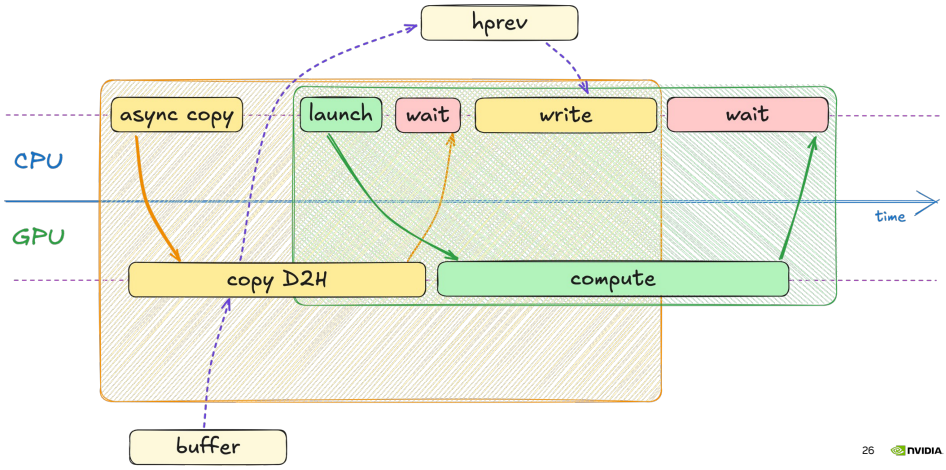

```
cudaError_t cudaMemcpyAsync(
  void*           dst,
  const void*     src,
  size_t        count,
  cudaMemcpyKind kind
);
```

25 NVIDIA

One subtlety of asynchronous operations is that errors don't always surface where they occur. For example, you could launch a kernel that goes out of bounds, but the GPU won't actually start executing it right away. Meanwhile, you might launch another kernel, that doesn't observe the error from the prior kernel, since it hasn't started running yet. But then cudaMemcpyAsync checks the GPU's status and discovers the error from the earlier kernel. In other words, an error triggered by any prior asynchronous operation might end up being reported by a completely different call. This is why it's crucial to check error codes after every CUDA call and be aware that an error can pop up much later than you'd expect.

CUDA Streams

- In this setup, `cudaMemcpyAsync` wouldn't give us any speedup
- All the submitted work **is ordered** on GPU



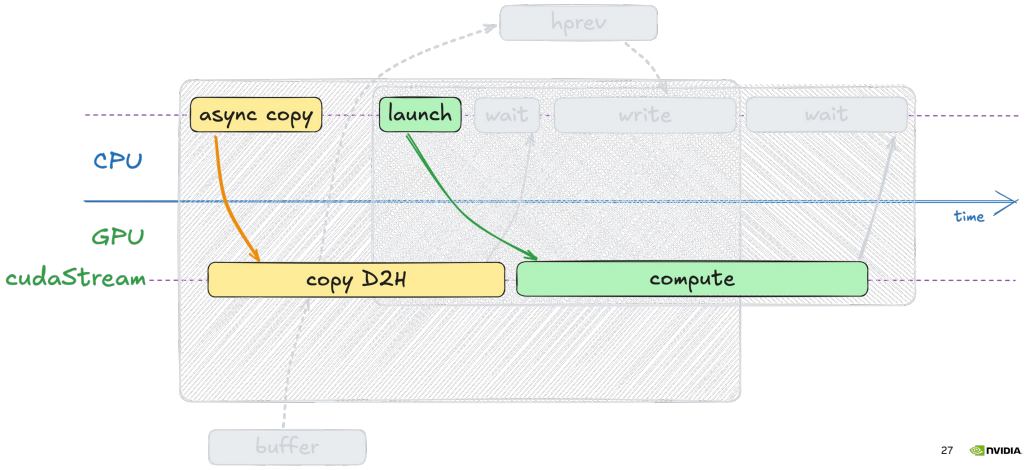
26 NVIDIA

Would simply swapping thrust::copy for cudaMemcpyAsync be enough to overlap compute with copy? Not really. By default, all GPU operations are ordered on the GPU end. Even though cudaMemcpyAsync is non-blocking on the CPU side, the GPU still executes it in order with everything else - just like those earlier asynchronous compute steps. This means that GPU will wait for copy to finish before proceeding to the next compute step.

NVIDIA

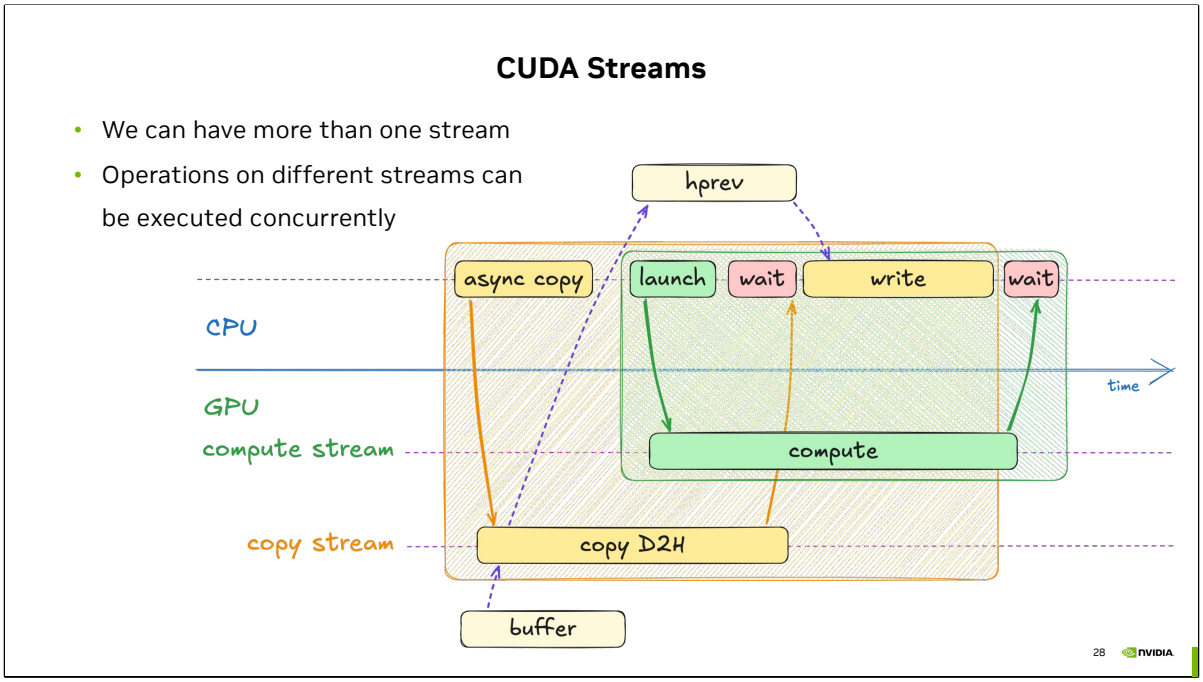
CUDA Streams

- This ordering of operations on GPU is called **cudaStream**
- When you don't specify a stream, default one is used



This ordering of operations on the GPU side is called a CUDA Stream. When you don't specify a CUDA stream, some default one is used.

NVIDIA



And the great news is that we can have as many CUDA streams as we need. When operations run in different streams, they are no longer ordered with respect to each other and can execute concurrently. That's exactly what we want: if we send an asynchronous copy to one stream and our compute work to another, then those two operations can overlap. Let's see how that works in practice.

Operations on Streams

```
cudaStream_t copy_stream, compute_stream;
```

I Construction

```
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);
```

II Destruction

```
cudaStreamDestroy(compute_stream);
cudaStreamDestroy(copy_stream);
```

III Synchronization

```
cudaStreamSynchronize(compute_stream);
cudaStreamSynchronize(copy_stream);
```

- Waits until all preceding commands in the stream have completed
- More lightweight compared to synchronizing the entire GPU

29 NVIDIA

cudaStream_t is simply a type, just like an integer. That means we can declare our streams the same way we'd declare any other variable in C++. In this example, we define two streams - one for memory transfers and another for computations.
However, just creating a stream variable isn't enough; we also need to explicitly call cudaStreamCreate to initialize it. Later, if we want to wait for the tasks in a specific stream to finish, we can use cudaStreamSynchronize, which blocks only until that particular stream's operations are complete. This is preferable to cudaDeviceSynchronize, which causes CPU to wait until *all* GPU operations have completed. By synchronizing at the stream level, we let other streams continue to work in parallel. Finally, once we're done with a stream, we should call cudaStreamDestroy to release associated resources.

Usage of CUDA Streams

CUDA Runtime

```
cudaError_t
cudaMemcpyAsync(
  void*           dst,
  const void*     src,
  size_t        count,
  cudaMemcpyKind kind,
  cudaStream_t stream = 0
);
```

CUB

```
cudaError_t
cub::DeviceTransform::Transform(
  IteratorIn    input,
  IteratorOut  output,
  int       num_items,
  TransformOp      op,
  cudaStream_t stream = 0
);
```

cuBLAS

```
cublasStatus_t
cublasLtMatmul(
  cublasLtHandle_t     lightHandle,
  cublasLtMatmulDesc_t computeDesc,
  const void                *alpha,
  const void                    *A,
  ...
  cudaStream_t              stream
);
```

- Majority of asynchronous CUDA libraries accept `cudaStream_t`
- The idea is that you'll likely want to overlap their API with:
  - memory transfers,
  - host-side compute or IO,
  - or even another device-side compute!

You'll find cudaStream_t in almost every accelerated CUDA library. For instance, both cudaMemcpyAsync and cub::DeviceTransform have a cudaStream_t parameter, though we haven't had to specify it yet because it defaults to zero, which means – use the default CUDA stream. The reason CUDA Streams is so widespread is that developers often want to overlap these library calls with other tasks. For example, you might run device-side computations in one stream while simultaneously handling host-side I/O or memory transfers in another,

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```

Create copy and compute streams

31 NVIDIA

So, let's start by creating CUDA streams for compute and copy operations.

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```

Copy device vector to host using asynchronous `cudaMemcpyAsync` on copy stream

After setting up a separate copy stream, we can pass it as the last argument to cudaMemcpyAsync so that the data transfer runs in its own stream.

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```

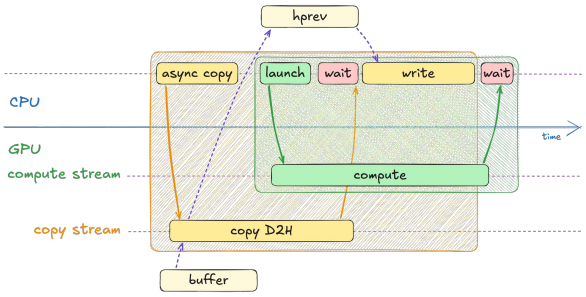
Launch CUB transform on compute stream

33 NVIDIA

Next, we can specify compute stream as the last parameter when calling cub::DeviceTransform. By doing so, we ensure the computation runs in a different stream than the copy operation.

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```



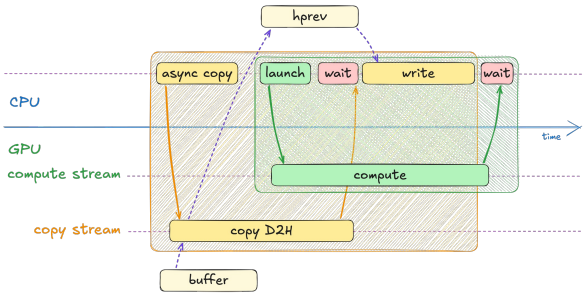
Synchronize copy stream before reading host vector

Because the copy happens asynchronously, we have to make sure it has actually finished before reading the host data. To do this, we call cudaStreamSynchronize on the copy stream, which blocks until operatiosn in the copy stream are complete.

NVIDIA

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```

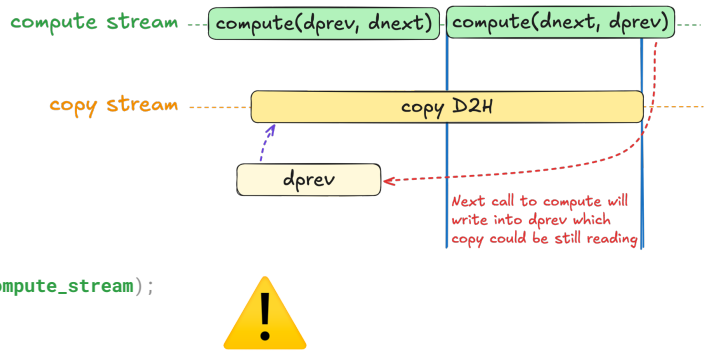


Synchronize compute stream before starting the next step

After we're done writing the results to disk, we synchronize with the compute stream to confirm that the GPU computations have also finished. In the end, our approach looks like the diagram on the slide: we perform two asynchronous operations - one for copying data, one for computing - in separate streams. We also include two synchronization points to ensure everything finishes in the right order.

CUDA Streams

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

// ...

cudaMemcpyAsync(hprev_ptr,
                dprev_ptr,
                num_cells * sizeof(float),
                cudaMemcpyDeviceToHost,
                copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);

store(write_step, height, width, hprev);
cudaStreamSynchronize(compute_stream);
```



- This code has a data race!
- Different streams execute their operations out of order

And… congratulations, we've just introduced our first data race! Let's walk through the first few iterations to see why: 1. We issue a copy from dprev to the host; 2. The first compute iteration reads from dprev and writes into dnext, which is fine; 3. But the next iteration reads from dnext and writes back into dprev. Because these operations run in different streams, the GPU can interleave them in unexpected ways. This means that while data is still being copied from dprev to the host, it might get overwritten by the next compute iteration in another stream. If we want correct visualization results, we'll have to fix this.

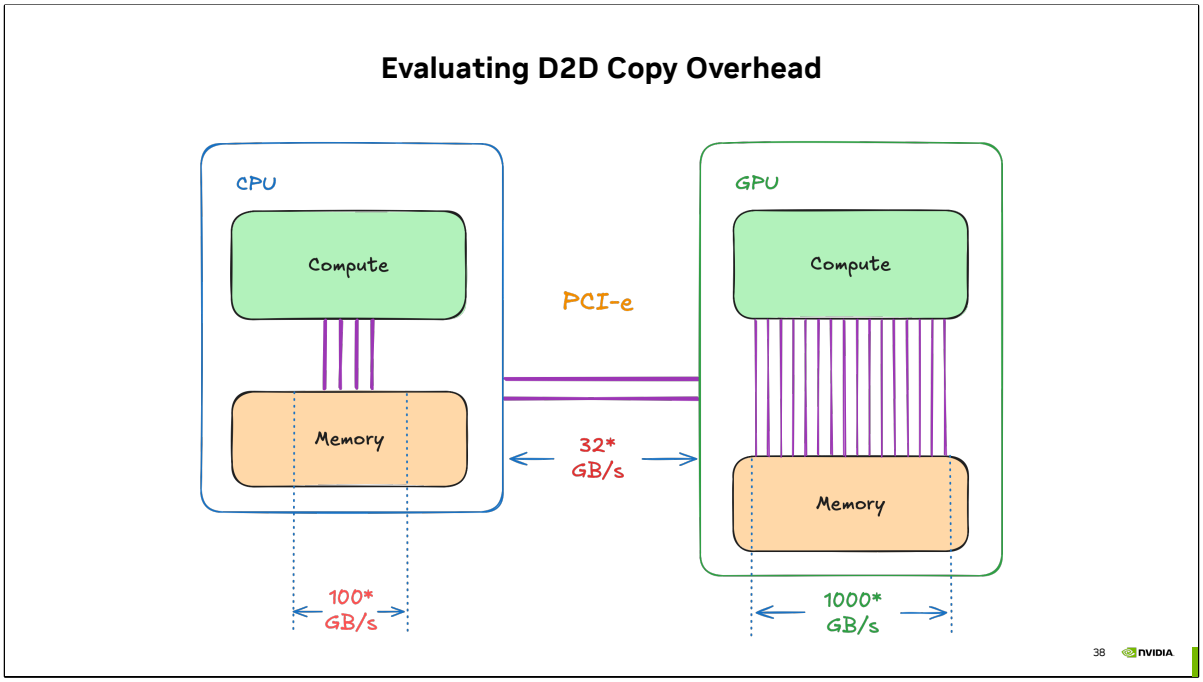
CUDA Streams

```
thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(htemp_ptr, dbuffer_ptr, num_bytes, cudaMemcpyDeviceToHost, copy_stream);

for (int step = 0; step < steps; step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}
```

- Most problems can be solved with another level of indirection
- Let's:
  - Allocate a device buffer
  - Copy dprev into it in compute stream
  - Start asynchronous copy from this buffer to hprev

37 NVIDIA

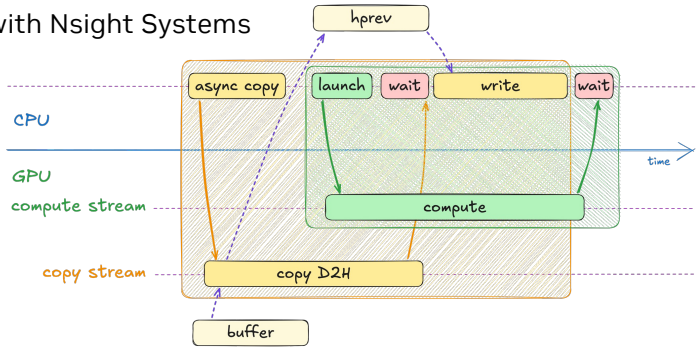
Like many computer science problems, we can solve this by adding another layer of indirection. We can allocate an extra device buffer, copy data into it synchronously, and then copy from that staging buffer to CPU. Since no one else is writing to the staging buffer, there's no risk of a data race. But doesn't that defeat the purpose of overlapping compute with transfers? At the first glance, it may seem so.



But if you take a closer look at the bandwidth across your entire system, this approach makes a lot more sense. Copying data within GPU memory can hit speeds of around a thousand gigabytes per second. Moving data within CPU memory is somewhat slower, on the order of a hundred gigabytes per second. However, transfers between the CPU and GPU often go through the PCIe bus, which might provide as little as 32 gigabytes per second. That means that the real bottleneck is cross-memory transfers between the CPU and GPU. The device-to-device copy is going to be so fast it's practically 'free' compared to the cost of sending data over PCIe. So, it still makes sense to overlap the slower device-to-host copy with ongoing GPU computations.

Exercise: Overlap Compute and Copy

- Use `cudaMemcpyAsync` instead of `thrust::copy`
- Put compute and async copy on different streams
- Profile resulting code with Nsight Systems



`02.03-Streams/02.03.02-Exercise-Async-Copy.ipynb`

That brings us to our next exercise. In this task, you'll replace thrust::copy with cudaMemcpyAsync to make your device-to-host transfer asynchronous. You'll also place your compute and copy operations on separate streams and synchronize them exactly as shown in the diagram on the slide. Finally, once you're comfortable with your changes, you'll profile the code in Nsight Systems to see if it behaves as expected.

NVIDIA

Exercise: Overlap Compute and Copy

Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::host_vector<float> hprev(height * width);

thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(thrust::raw_pointer_cast(htemp.data()),
                thrust::raw_pointer_cast(dbuffer.data()),
                num_cells * sizeof(float), cudaMemcpyDeviceToHost,
                copy_stream);

for (int compute_step = 0; compute_step < compute_steps; compute_step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);
store(write_step, height, width, hprev);

cudaStreamSynchronize(compute_stream);
```

Create compute and copy streams

40 NVIDIA

Let’s walk through the solution. The first step is to create two CUDA streams - compute-stream for GPU computations and copy-stream for data transfers. We do this by declaring two cudaStream_t variables and calling cudaStreamCreate on them.

Exercise: Overlap Compute and Copy

Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::host_vector<float> hprev(height * width);

thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(thrust::raw_pointer_cast(htemp.data()),
                thrust::raw_pointer_cast(dbuffer.data()),
                num_cells * sizeof(float), cudaMemcpyDeviceToHost,
                copy_stream);

for (int compute_step = 0; compute_step < compute_steps; compute_step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);
store(write_step, height, width, hprev);

cudaStreamSynchronize(compute_stream);
```

Synchronously copy into the staging buffer

41 NVIDIA

Next, we perform a device-to-device copy from our previous temperatures into a staging buffer. We can still use thrust::copy here, because device-to-device copies are generally very. This step ensures our device data we copy to host remains untouched by subsequent computations, preventing any data races.

Exercise: Overlap Compute and Copy

Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::host_vector<float> hprev(height * width);

thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(thrust::raw_pointer_cast(htemp.data()),
                thrust::raw_pointer_cast(dbuffer.data()),
                num_cells * sizeof(float), cudaMemcpyDeviceToHost,
                copy_stream);

for (int compute_step = 0; compute_step < compute_steps; compute_step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);
store(write_step, height, width, hprev);

cudaStreamSynchronize(compute_stream);
```

Asynchronously copy from staging buffer into host vector in the copy stream

After that, we invoke the asynchronous copy on the copy stream by calling cudaMemcpyAsync. Since cudaMemcpyAsync operates on raw pointers, we use thrust::raw_pointer_cast to retrieve the underlying pointers from Thrust containers.

Exercise: Overlap Compute and Copy

Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::host_vector<float> hprev(height * width);

thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(thrust::raw_pointer_cast(htemp.data()),
                thrust::raw_pointer_cast(dbuffer.data()),
                num_cells * sizeof(float), cudaMemcpyDeviceToHost,
                copy_stream);

for (int compute_step = 0; compute_step < compute_steps; compute_step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);
store(write_step, height, width, hprev);

cudaStreamSynchronize(compute_stream);
```

Launch compute on compute stream

43 NVIDIA

Finally, we update the CUB transform call to run on the compute stream we created earlier. By specifying the stream as the last parameter, we're telling CUB to execute this transformation independently of any operations on other streams.

NVIDIA

Exercise: Overlap Compute and Copy

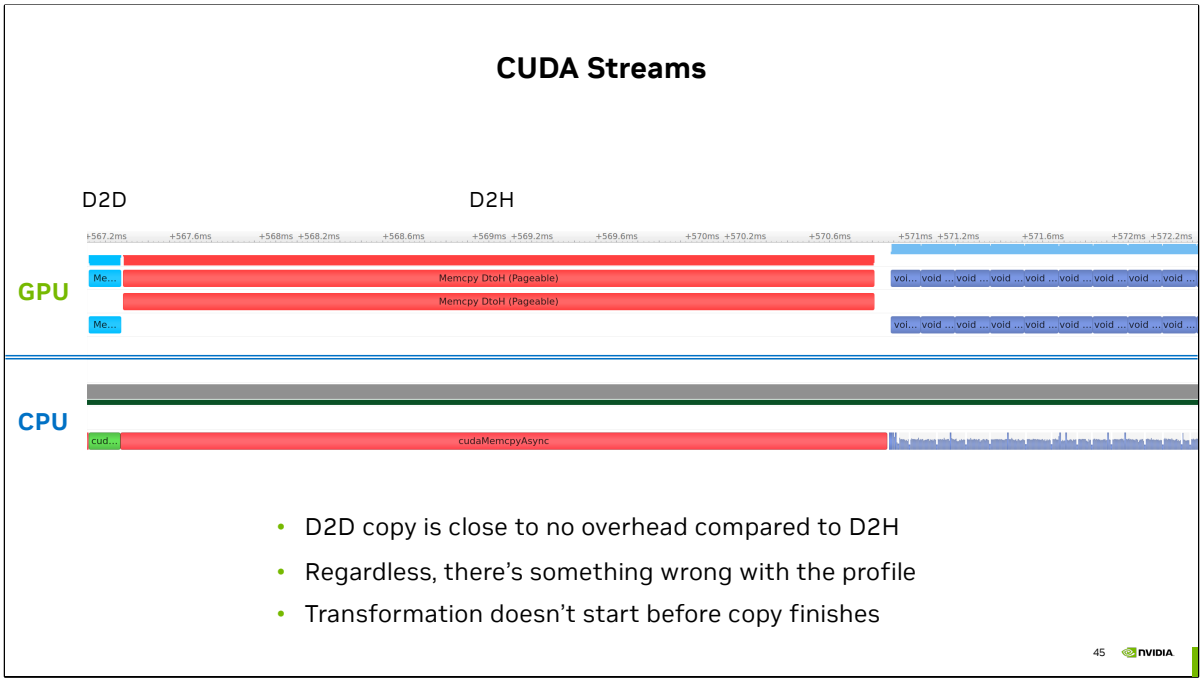
Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::host_vector<float> hprev(height * width);

thrust::copy(d_prev.begin(), d_prev.end(), d_buffer.begin());
cudaMemcpyAsync(thrust::raw_pointer_cast(htemp.data()),
                thrust::raw_pointer_cast(dbuffer.data()),
                num_cells * sizeof(float), cudaMemcpyDeviceToHost,
                copy_stream);

for (int compute_step = 0; compute_step < compute_steps; compute_step++)
{
  simulate(width, height, dprev, dnext, compute_stream);
  dprev.swap(dnext);
}

cudaStreamSynchronize(copy_stream);
store(write_step, height, width, hprev);

cudaStreamSynchronize(compute_stream);
```

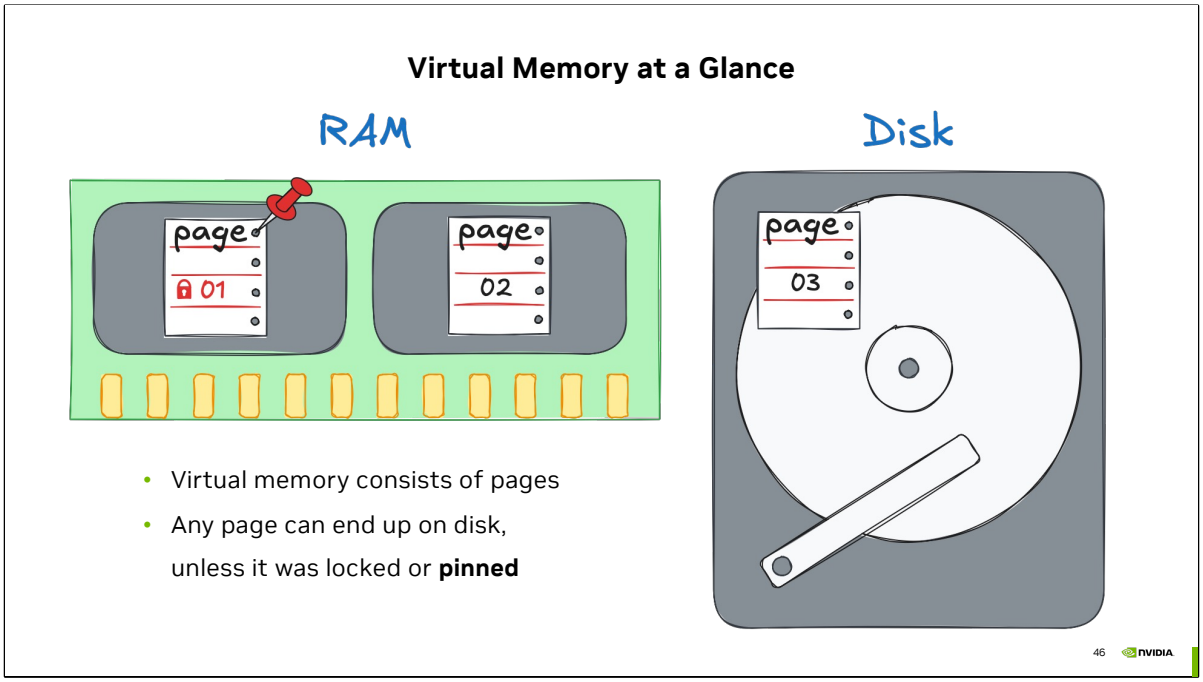
Wait for copy on the copy stream to finish before reading the data

44 NVIDIA

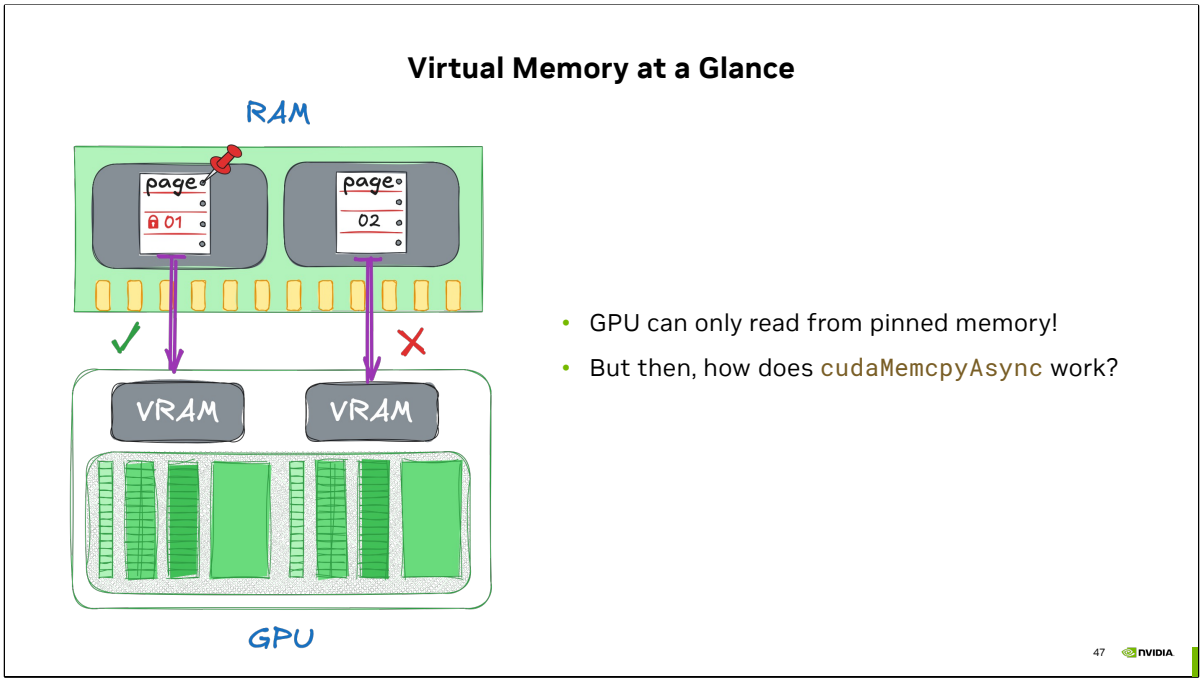
After launching all the asynchronous operations, we call cudaStreamSynchronize on the copy stream to make sure the data transfer to the host has finished. Only once this function returns do we know that the host data is safe to read. At that point, we can proceed to writing the updated host data to disk.



After looking at the Nsight profile, we have both good news and bad news. The good news is that our intuition about the cost of data transfers was correct! The device-to-device copy on the left is significantly faster than the device-to-host copy in the middle. This confirms that the PCIe transfer is the real bottleneck. The bad news is that, despite using separate streams, compute is still waiting for the copy to finish. That seems puzzling - didn't we just separate them into different streams to avoid this serialization? It turns out this isn't entirely bad news. It means that there's more to learn about how the GPU hardware works.

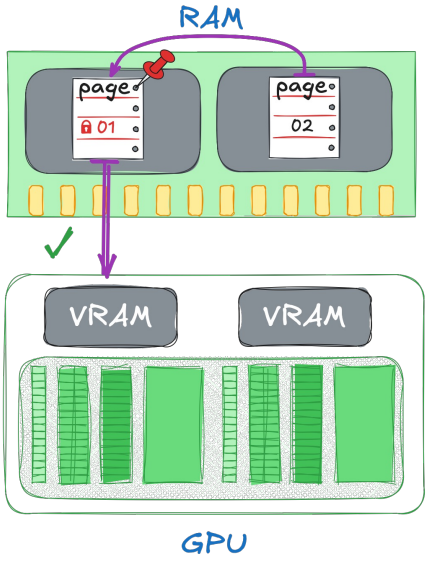


To understand why our copy and compute operations didn't overlap, we need to look at how memory actually works. Your program uses virtual memory, which is divided into pages. Most of the time, these pages are backed by physical RAM, but if RAM is short on space, the operating system can swap pages out to disk. As a result, at any given moment, you can't be entirely certain whether a particular page is in RAM or on disk. To make sure a page remains in physical memory, we can 'page-lock' or 'pin' it. By pinning pages, we tell the operating system it's not allowed to move them to disk.



So how does this relate to GPUs? As it turns out, a GPU can only read from pinned (page-locked) memory. That raises a question, though: we haven't done anything special when transferring data between the CPU and GPU. So, how did it work up until now?

Virtual Memory at a Glance



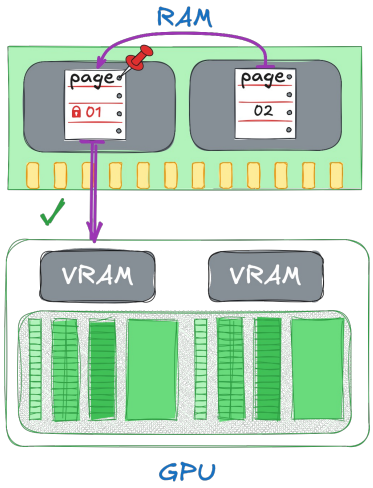
- Transfers from pageable memory work through a staging buffer in pinned memory
- Hence, transfers from pageable memory take longer and hinders asynchrony
- Fortunately, we can use pinned memory with `thrust::universal_host_pinned_vector`

48 NVIDIA

Behind the scenes, the CUDA runtime uses a small pinned buffer as a staging area for transfers. When we copy data from regular (pageable) memory, the driver moves a chunk of data into this staging buffer, sends that chunk to the GPU, waits for it to finish, and then repeats. This process effectively makes the copy synchronous, blocking any chance to overlap with computation. The good news is we can avoid this by allocating pinned memory ourselves. In Thrust, this is as simple as using a universal host pinned vector instead of a regular host vector. By doing so, we bypass the hidden staging step and enable truly asynchronous transfers that can overlap with our GPU workloads.

NVIDIA

Exercise: Use Pinned Memory



- Use `thrust::universal_host_pinned_vector` to allocate host vector in pinned memory
- Profile your application after the change
- Locate device-to-host copy and identify if it is being overlapped with compute

`02.04-Pinned-Memory/02.04.02-Exercise-Copy-Overlap.ipynb`

49 NVIDIA

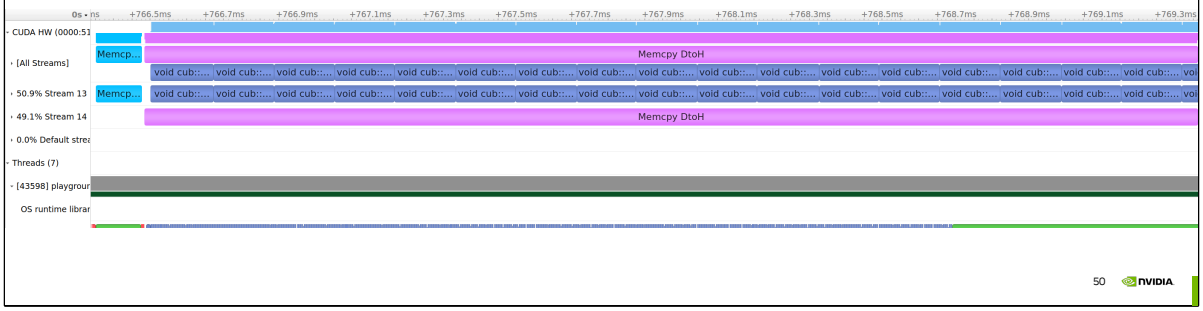
Which brings us to the final exercise in this section. This time, you'll allocate pinned memory by using a universal host pinned vector instead of a regular host vector. After making that change, run your application under Nsight Systems again and look for the device-to-host copy in the timeline. See if it now overlaps with compute.

NVIDIA

Exercise: Use Pinned Memory

Solution

```
cudaStream_t copy_stream, compute_stream;
cudaStreamCreate(&compute_stream);
cudaStreamCreate(&copy_stream);

thrust::universal_host_pinned_vector<float> hprev(height * width);

cudaMemcpy(thrust::raw_pointer_cast(dbuffer.data()),
           thrust::raw_pointer_cast(dprev.data()),
           num_cells * sizeof(float), cudaMemcpyDeviceToDevice);
...
```



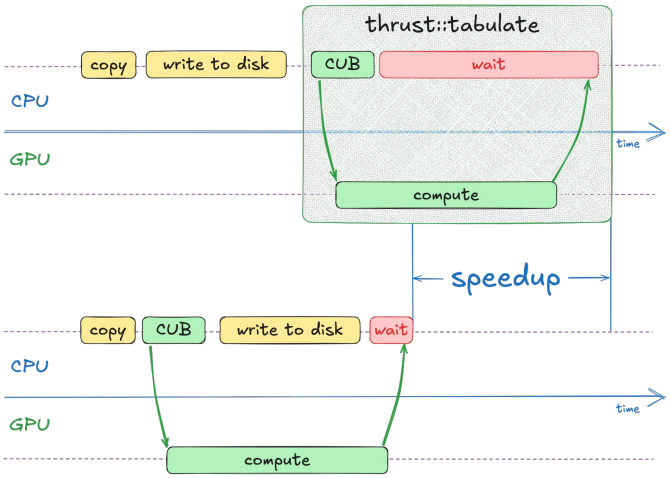
The change is pretty straightforward - just switch your host vector to the universal host pinned vector. With that done, the device-to-host transfer now overlaps compute. As the profiling timeline shows, GPU can perform many compute steps while data is being copied. Which further improves efficiency of our program.

NVIDIA

Takeaways

Prefer asynchronous programming model:

- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations



51 NVIDIA

Let's quickly go over some key takeaways. Prefer asynchronous programming model to overlap memory transfers, CPU tasks, and GPU computations.

NVIDIA

Takeaways

Prefer asynchronous programming model:

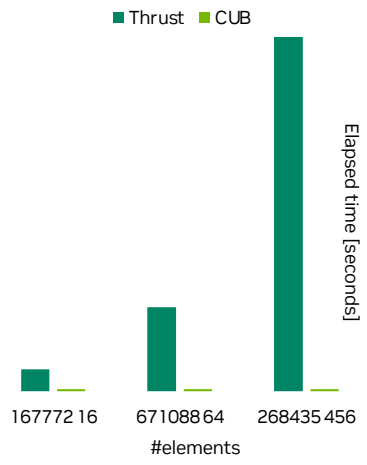
- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations

Use CUB for

- asynchronous general-purpose parallel algorithms

CUB

```
auto begin = std::chrono::high_resolution_clock::now();
auto cell_ids = thrust::make_counting_iterator(0);
cub::DeviceTransform::Transform(
  cell_ids, out.begin(), num_cells, compute);
auto end = std::chrono::high_resolution_clock::now();
```



52 NVIDIA

Use CUB for general-purpose asynchronous parallel algorithms

NVIDIA

Takeaways

Prefer asynchronous programming model:

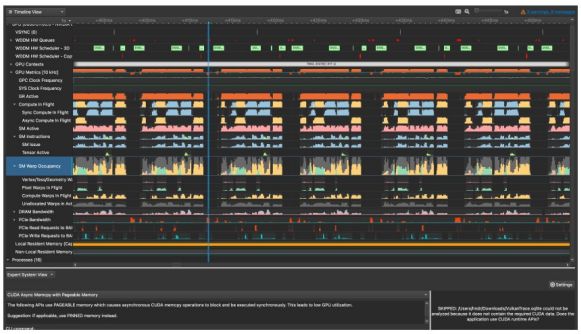
- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations

Use CUB for

- asynchronous general-purpose parallel algorithms

Use NVIDIA Nsight Systems:

- to profile your code
- identify opportunities for optimization
- visualize asynchronous execution
- use NVTX to simplify profiling



53 NVIDIA

Use Nsight to profile your code, identify opportunities for optimization, and visualize asynchronous execution. To further simplify profiling with Nsight, use NVTX ranges.

NVIDIA

Takeaways

Prefer asynchronous programming model:

- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations

Use CUB for

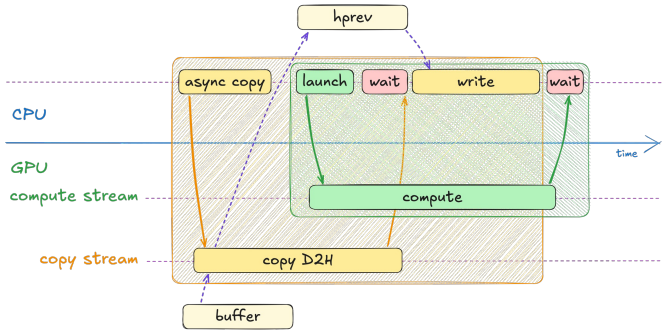
- asynchronous general-purpose parallel algorithms

Use NVIDIA Nsight Systems:

- to profile your code
- identify opportunities for optimization
- visualize asynchronous execution
- use NVTX to simplify profiling

Use `cudaStream_t`:

- to enable asynchronous execution with CPU and other streams



54 NVIDIA

Use CUDA streams to enable asynchronous execution with CPU and other streams

NVIDIA

Takeaways

Prefer asynchronous programming model:

- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations

Use CUB for

- asynchronous general-purpose parallel algorithms

Use NVIDIA Nsight Systems:

- to profile your code
- identify opportunities for optimization
- visualize asynchronous execution
- use NVTX to simplify profiling

Use `cudaStream_t`:

- to enable asynchronous execution with CPU and other streams

Use `cudaMemcpyAsync`:

- to make memory copies asynchronous with respect to CPU

```
cudaError_t cudaMemcpyAsync(
  void*           dst,   → Destination memory address
  const void*     src,   → Source memory address
  size_t          count, → Size in bytes to copy
  cudaMemcpyKind kind    → cudaMemcpyHostToHost
                         → ...
);
```

55 NVIDIA

Use cudaMemcpyAsync to make copies asynchronous with respect to CPU

Takeaways

Prefer asynchronous programming model:

- to overlap memory transfers
- to overlap CPU and GPU computation
- or even different GPU computations

Use CUB for

- asynchronous general-purpose parallel algorithms

Use NVIDIA Nsight Systems:

- to profile your code
- identify opportunities for optimization
- visualize asynchronous execution
- use NVTX to simplify profiling

Use `cudaStream_t`:

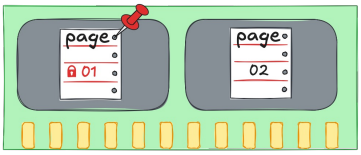
- to enable asynchronous execution with CPU and other streams

Use `cudaMemcpyAsync`:

- to make memory copies asynchronous with respect to CPU

Use pinned memory to:

- to make copy from and to device asynchronous
- make copy from and to device faster



56 NVIDIA

Finally, use pinned memory for asynchronous data transfers and improved performance

NVIDIA