Unlocking the GPU’s Full Potential: Asynchrony and CUDA Streams

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

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

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?

Data Dependency

- Write step reads host copy of previous state
- Compute step reads device copy of previous state
- So, there's no dependency between the two
- Meaning that **CPU** can spend time writing results on disk

- But that'd require splitting Thrust call into launch and wait

CUB

Sorting

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

Iterators

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

Types

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

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

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`

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

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

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

Exercise: Compute-IO Overlap

10 minutes

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

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);
  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);
}
```

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);
}
```

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

Exercise: Profile Your Code with Nsight Systems

10 minutes

`02.02-Asynchrony/02.02.03-Exercise-Nsight.ipynb`

Exercise: Profile Your Code with Nsight Systems

10 minutes

File Edit View Run Kernel Tabs NVIDIA Nsight Settings Help

/ 02.02-Asynchrony /

Name	Modified
Solutions	5 seconds ago
Sources	5 seconds ago
02.02.01-Asynchro...	5 seconds ago
02.02.02-Exercise...	5 seconds ago
02.02.03-Nsight.ip...	now

Launcher | 02.02.03-Nsight.ipynb

Code | Python 3 (ipykernel)

[1]:
```
!nvcc --extended-lambda -o /tmp/a.out Solutions/compute-io-overlap.cu # build executable
!nsys profile -o ../nsight-reports/compute-io-overlap /tmp/a.out # run and profile executable
```

```
Collecting data...
compute + write 0 in 0.144594 s
          write 0 in 0.0377519 s
compute + write 1 in 0.144425 s
          write 1 in 0.0368032 s
compute + write 2 in 0.144422 s
          write 2 in 0.0371233 s
Generating '/tmp/nsys-report-1fa3.qdstrm'
[1/1] [========================100%] compute-io-overlap.nsys-rep
Generated:
    /dli/task/02.02-Asynchrony/../nsight-reports/compute-io-overlap.nsys-rep
```

[ ]:

Profile your app with `nsys profile` and store the report file

Exercise: Profile Your Code with Nsight Systems

Click "Connect to Nsight Tool UI" in the "NVIDIA Nsight" tab

Exercise: Profile Your Code with Nsight Systems

Open Nsight Systems: File – Open…

Exercise: Profile Your Code with Nsight Systems

Navigate to the report you just saved

Exercise: Profile Your Code with Nsight Systems

Click and drag to select a time region

Exercise: Profile Your Code with Nsight Systems

Click “Filter and Zoom in”

Exercise: Profile Your Code with Nsight Systems

Unfold “CUDA HW” to see more details on what GPU is doing

Exercise: Profile Your Code with Nsight Systems

10 minutes

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`

Exercise: Profile code with Nsight

Solution

Exercise: Profile code with Nsight

Solution

- Algorithms are now asynchronous
- Synchronization with GPU is now explicit

Exercise: Profile code with Nsight

- Algorithms are now asynchronous
- Synchronization with GPU is now explicit
- Compute on GPU now overlaps writing on CPU

Synchronous vs Asynchronous

Synchronous

Asynchronous

- Our simulation has enough compute to fully overlap IO
- This makes version with compute / IO overlap almost twice as fast

NVIDIA Tools Extension (NVTX)

https://github.com/NVIDIA/NVTX

- Inspecting Nsight report for a complex application can be intimidating
- NVTX allows you to create custom ranges in Nsight Systems from code

```
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)};
    ...
  }
}
```

Exercise: Use NVTX

8 minutes

- 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

Exercise: Use NVTX

Solution

- Now all the steps are much easier to locate

CUDA Streams

- Observe how compute unnecessarily waits for copy to finish
- Fortunately, there's `cudaMemcpyAsync`, that can make copy asynchronous as well

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,
  const void*      src,
  size_t         count,
  cudaMemcpyKind kind
);
```

- `dst` → Destination memory address
- `src` → Source memory address
- `count` → Size **in bytes** to copy
- `kind` →
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

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
);
```

CUDA Streams

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

CUDA Streams

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

CUDA Streams

- We can have more than one stream
- Operations on different streams can be executed concurrently

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

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!

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

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

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

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

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

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

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

Evaluating D2D Copy Overhead

Exercise: Overlap Compute and Copy

15 minutes

- 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`

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

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

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

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

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

CUDA Streams

- D2D copy is close to no overhead compared to D2H
- Regardless, there’s something wrong with the profile
- Transformation doesn’t start before copy finishes

Virtual Memory at a Glance

- Virtual memory consists of pages
- Any page can end up on disk, unless it was locked or **pinned**

Virtual Memory at a Glance

- GPU can only read from pinned memory!
- But then, how does `cudaMemcpyAsync` work?

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`

Exercise: Use Pinned Memory

5 minutes

- 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`

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);
...
```

Takeaways

Prefer asynchronous programming model:

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

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();
```

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

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

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,
  const void*     src,
  size_t          count,
  cudaMemcpyKind  kind
);
```

- `dst` → Destination memory address
- `src` → Source memory address
- `count` → Size **in bytes** to copy
- `kind` → `cudaMemcpyHostToHost`, `...`

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

Break

10 minutes