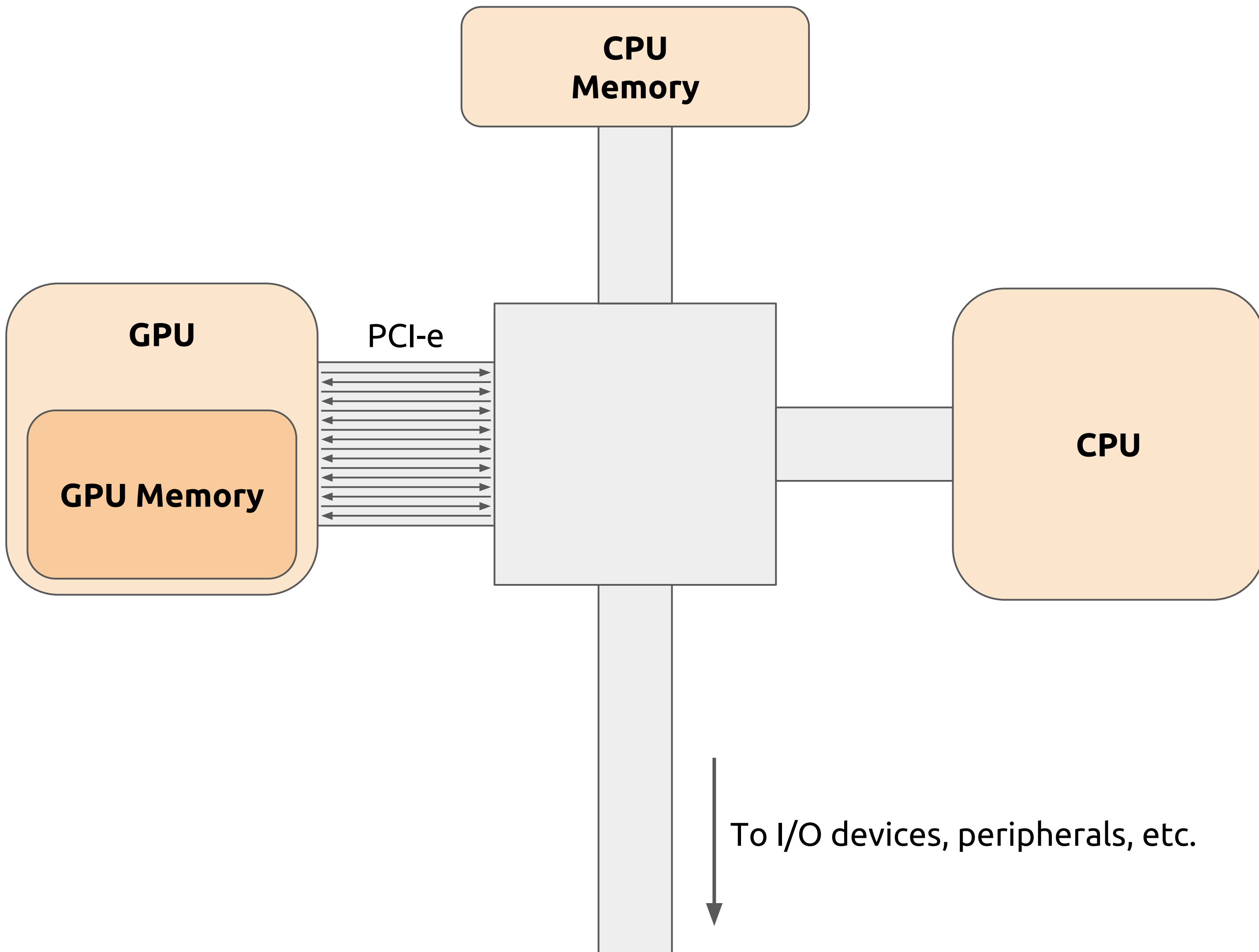
CSE 599 I
Accelerated Computing - Programming GPUS

Advanced Host / Device Interface

Objective

- Take a slightly lower-level view of the CPU / GPU interface
- Learn about different CPU / GPU communication techniques

A Prototypical High-Level PC Architecture



Peripheral Component Interconnect Express (PCI-e)

A high-throughput serial expansion bus

Transfers data over 1, 2, 4, 8, 16, or 32 (GPUs generally use PCI-e x16)

PCI-e Version	Year	x16 Throughput
1.0	2003	4 GB / s
2.0	2007	8 GB / s
3.0	2010	15.8 GB / s
4.0	2017 (expected)	31.5 GB / s

GTX 1080 Ti

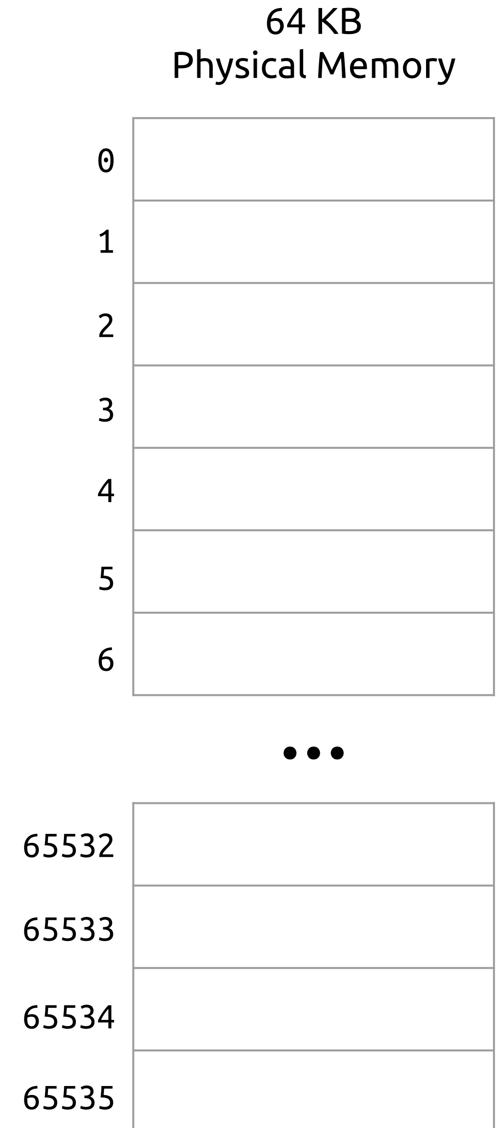
PCI-e 3.0 x16
cudaMemcpy comes through here!

Review: Physical Addressing

- Physical addressing assigns consecutive numbers (i.e. addresses) to consecutive memory locations, from 0 to the size of the memory

- One byte addressing is typical

- In the memory shown, bytes are assigned addresses 0 - 65535



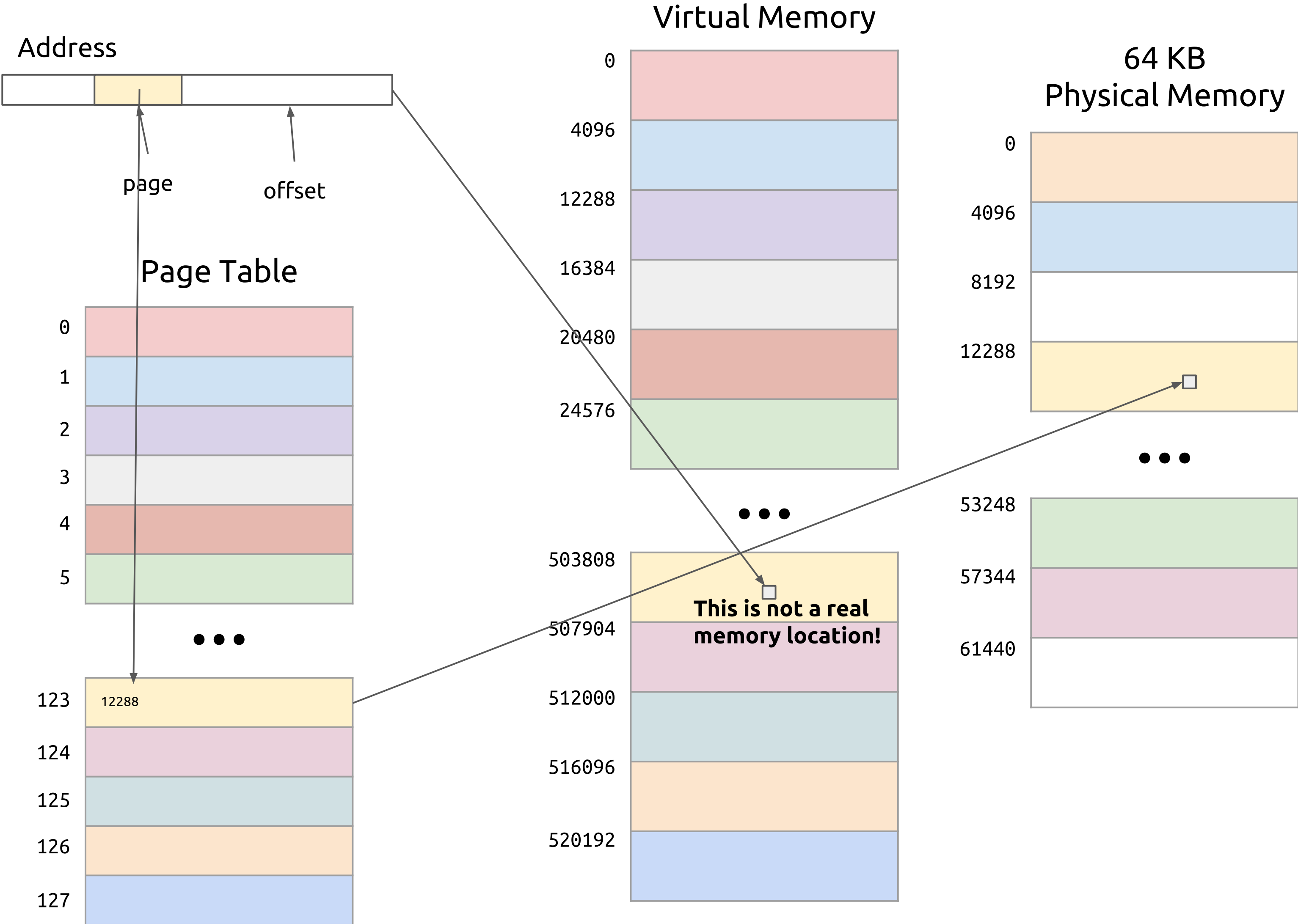
Review: Virtual Addressing

A virtual address points into a virtual address space that does not have a fixed 1-1 mapping to physical memory

Virtual address spaces can be much larger than available physical memory

Virtual addresses are typically divided into pages of at least 4096 bytes

Review: Virtual Addressing



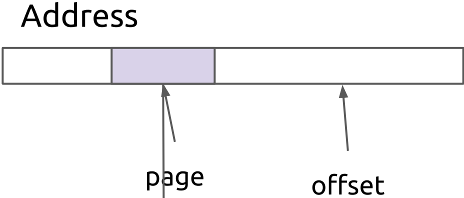
Why?

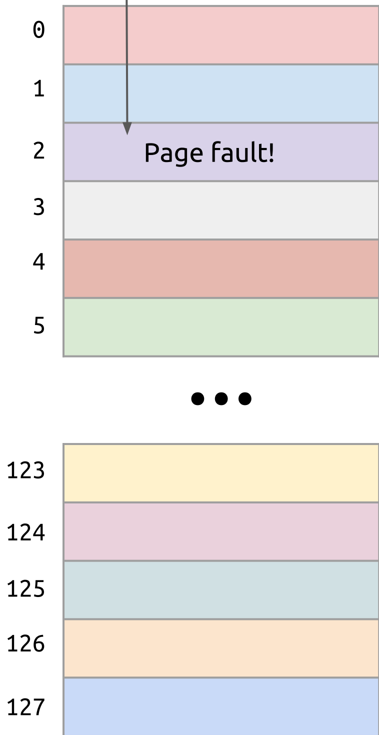
Can support larger memories than are physically available

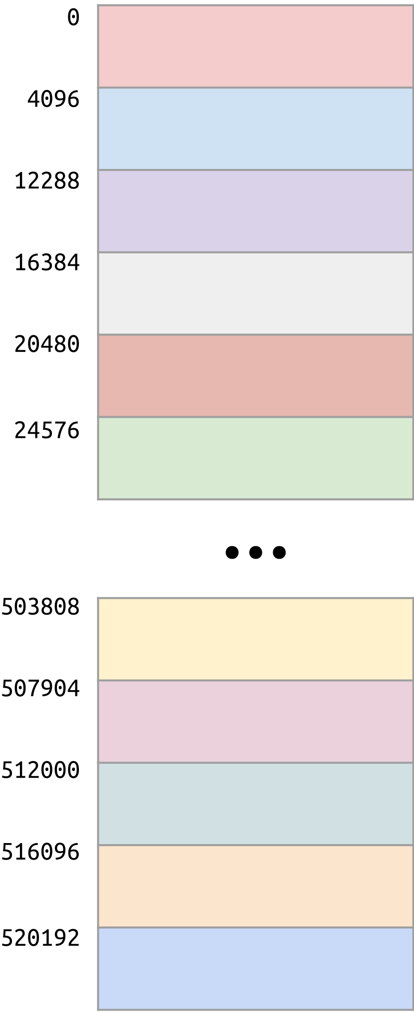
Page-level permissions information makes “sandboxing” of separate processes easier

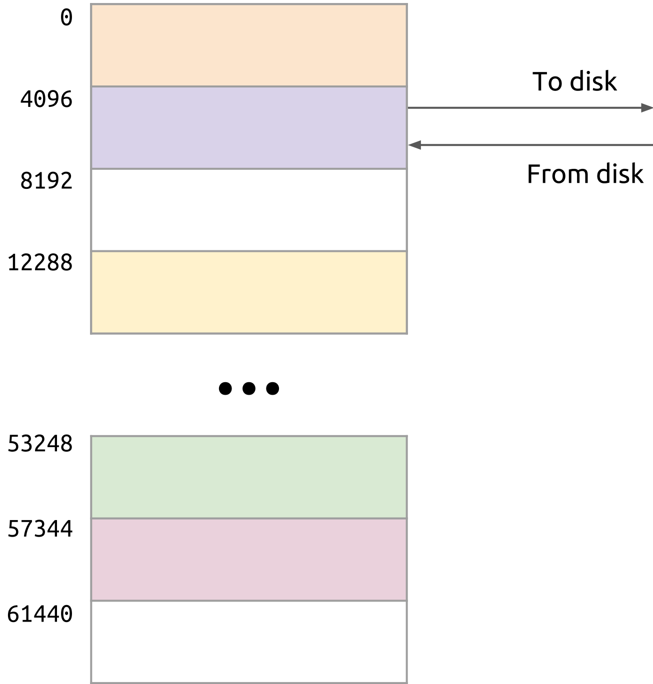
Multiple physical memories, files, and I/O devices can be mapped into a virtual memory space

Review: Demand Paging





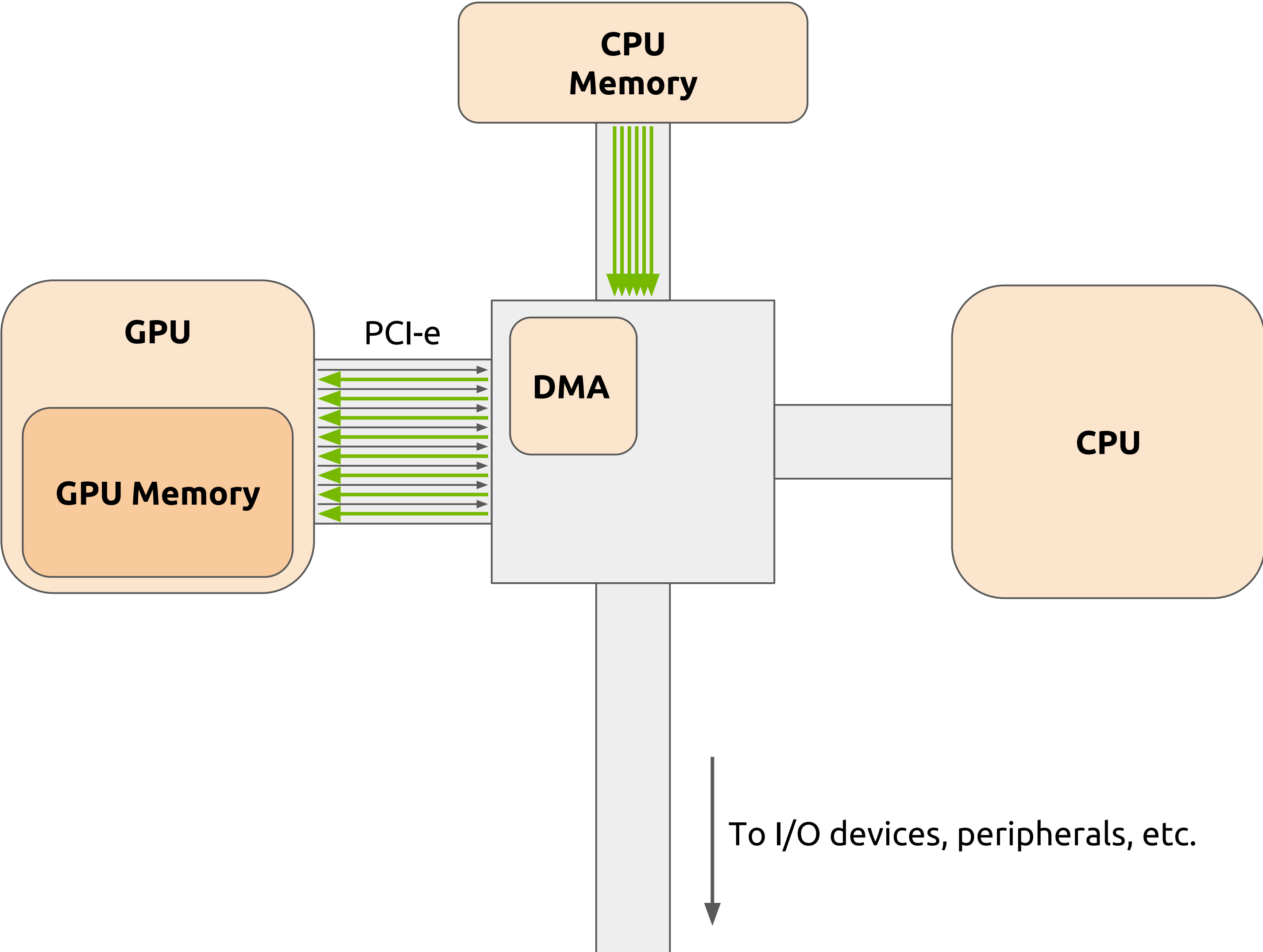




CUDA Virtual Addressing

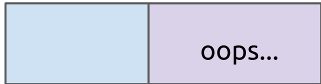
- CUDA supports virtual addressing such that CUDA processes cannot "accidentally" read each others' data
- CUDA does **not** support demand paging
    - every byte of virtual memory must be backed up by a byte of physical memory
- Traditionally, CUDA and CPU address spaces were completely separate

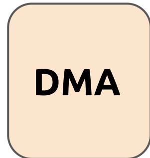
Direct Memory Access (DMA)

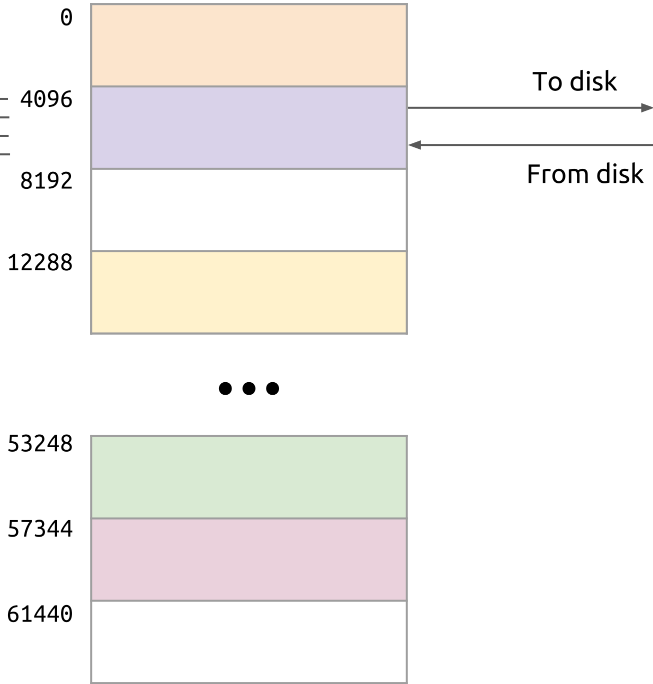


DMA Danger

Section of Device Memory








```
cudaMemcpy( [ ], [ ], N, cudaMemcpyHostToDevice);
```

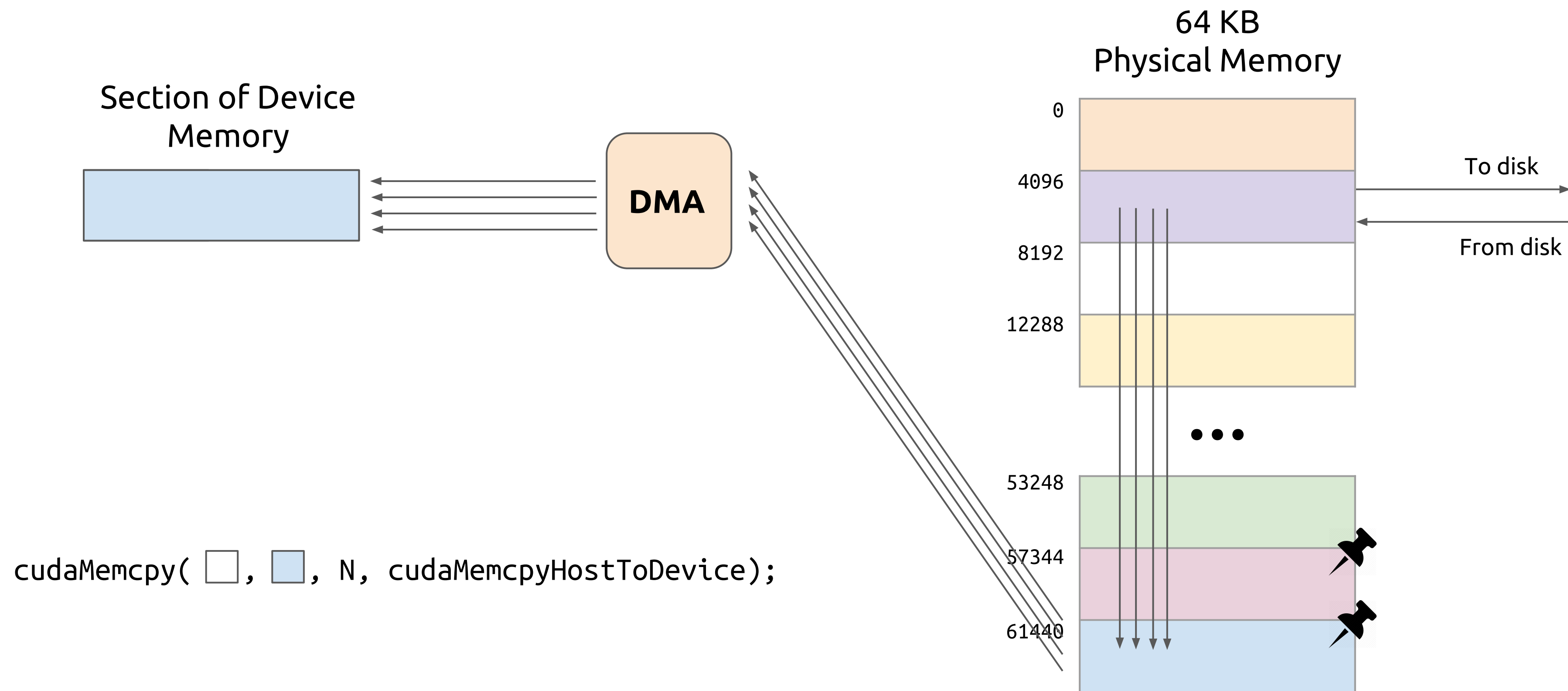
Pinned Memory

Pinned memory (akak page-locked memory) is a part of the virtual address space that is “pinned” to a part of the physical space

Pinned memory cannot be “paged out”

The DMA uses pinned memory for data transfer

cudaMemcpy Uses Pinned Memory



Cutting out the Middleman

You can allocate a host buffer in pinned memory directly:

```
cudaHostAlloc(&hBuffer, size, cudaHostAllocDefault);
...
cudaMemcpy(dBuffer, hBuffer, size, cudaMemcpyHostToDevice);
...
cudaHostFree(hBuffer);
```

This eliminates the extra copy to pinned memory

Keep in mind that pinned memory is a limited resource

Zero-Copy Memory

Introduced as part of CUDA 2.2 in 2009

Allows mapping pinned host memory into the CUDA address space

Values are copied from the pinned memory to the device as they are requested

There is no caching

Writing simultaneously to zero-copy memory from both host and device leads to undefined behavior

Zero-Copy Memory Setup

```
cudaSetDeviceFlags(cudaDeviceMapHost);

cudaSetDevice( deviceNumber );

cudaDeviceProp deviceProperties;
cudaGetDeviceProperties(&deviceProperties, deviceNumber);

if (!deviceProperties.canMapHostMemory) {
     printf("Device %d does not support mapped memory\n");
     exit(EXIT_FAILURE);
}
```

Vector Addition with Zero-Copy Memory

```
cudaHostAlloc(&h_A, N * sizeof(float), cudaHostAllocMapped);
cudaHostAlloc(&h_B, N * sizeof(float), cudaHostAllocMapped);

// copy input data to h_A and h_B
...

cudaHostGetDevicePointer(&d_A, h_A, 0);
cudaHostGetDevicePointer(&d_B, h_B, 0);
cudaMalloc(&d_C, N * sizeof(float));

// set up grid
...

vectorAdditionKernel<<<grid, block>>>(d_A, d_B, d_C, N);


// use results in d_C
...

cudaFreeHost(h_A);
cudaFreeHost(h_B);
cudaFree(d_C);
```

An Alternative: Register Pre-Allocated Memory

One can also page-lock an existing buffer:

```
float * h_A;

cudaHostRegister(h_A, N * sizeof(float), cudaHostRegisterDefault);

...

cudaHostUnregister(h_A);
```

Zero-Copy Memory Tradeoffs

- Simplifies code by removing explicit data transfer
- Allows dynamic overlap of transfer and computation
- Can be extremely efficient for integrated graphics
- Can be used to process more data than fits in device memory

- No caching
- Must synchronize memory writes with the host
- Overuse of pinned memory can slow down the host

When to Use Zero-Copy Memory on Discrete GPUs

- When the data will be read or written only once
- When the reads will all be coalesced

Unified Virtual Addressing (UVA)

Introduced as part of CUDA 4 in 2011

Supported on 64-bit systems and devices with compute capability at least 2.0

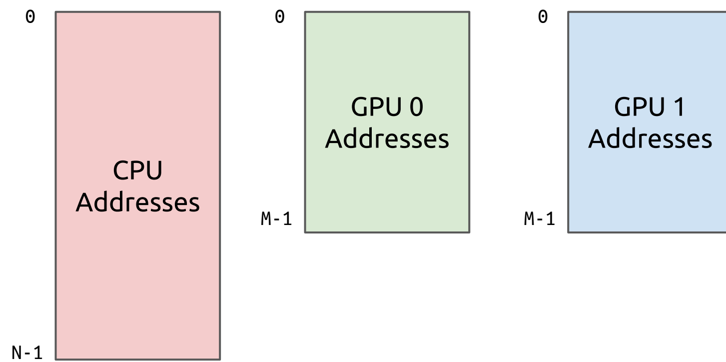
All addresses are in the same address space

The runtime API call `cudaPointerGetAttributes()` can be used to determine whether a pointer points to host or device memory

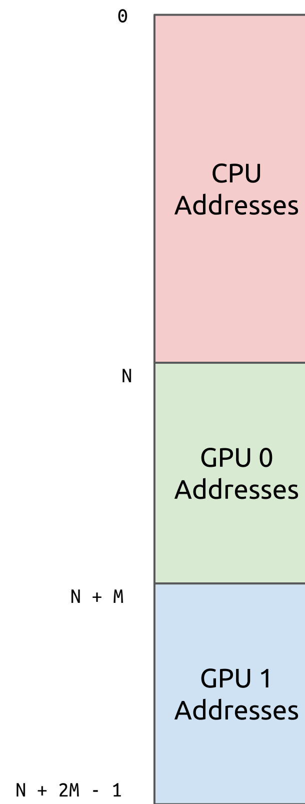
You still can't dereference a host pointer on the device or vice versa (unless it's mapped memory)

Unified Virtual Addressing (UVA)

Before UVA



UVA



Unified Memory

Introduced as part of CUDA 6 in 2013

Dependent on Unified Virtual Addressing, but not the same thing

Extends UVA with the ability to have *managed* memory, which is automatically migrated between host and device

Vector Addition With Unified Memory

```
cudaMallocManaged(&A, N * sizeof(float));
cudaMallocManaged(&B, N * sizeof(float));
cudaMallocManaged(&C, N * sizeof(float));

// copy input data to h_A and h_B
...

// set up grid
...

vectorAdditionKernel<<<grid, block>>>(A, B, C, N);

// C is able to be dereferenced from the host
...

cudaFree(A);
cudaFree(B);
cudaFree(C);
```

Managed Static Memory

Static memory can also be managed:

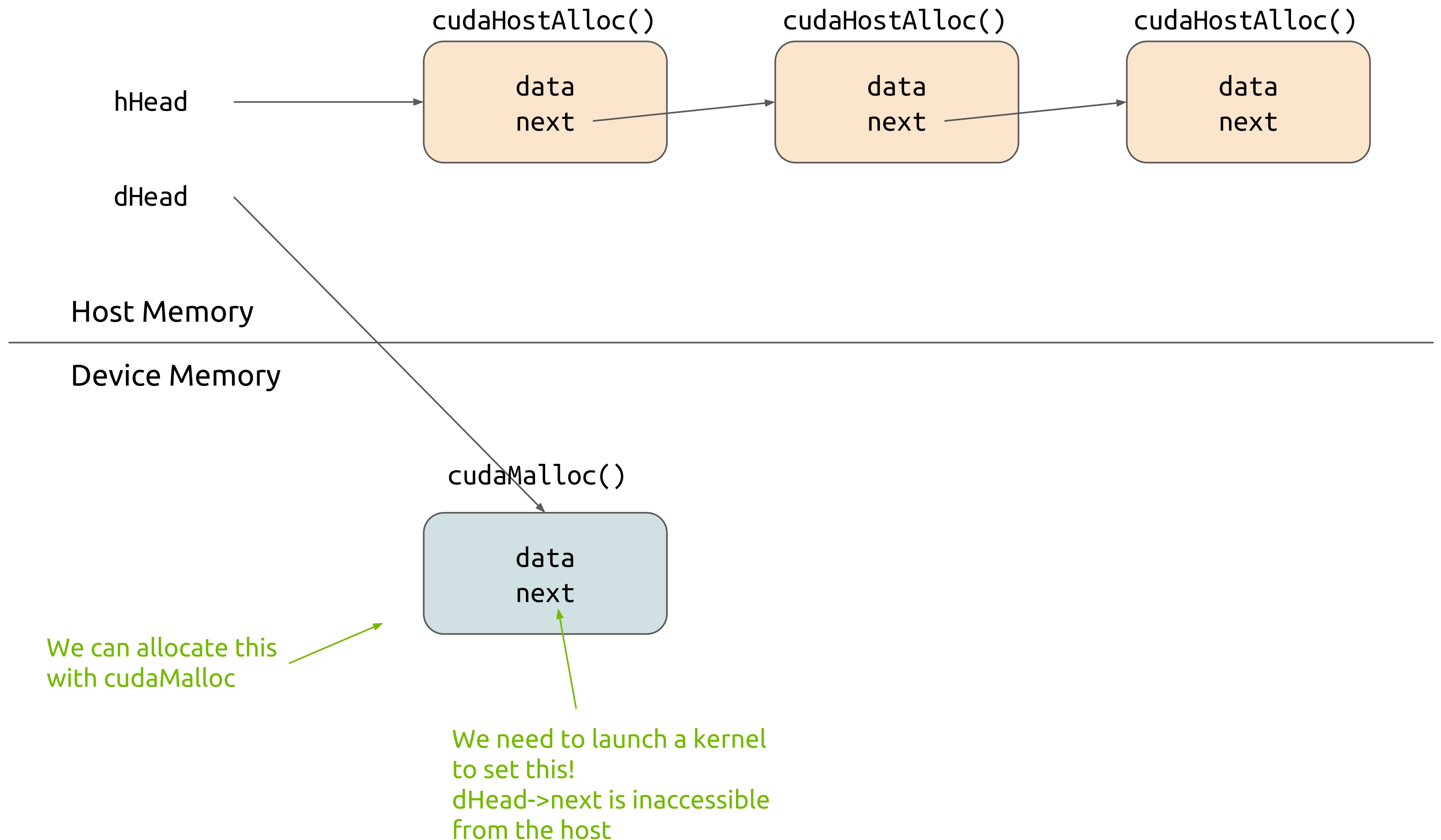
```
__device__ __managed__ int sum;
```

Can be useful for seamlessly transferring results back from, e.g., a reduction

Advantages of Unified Memory

- Much simpler code
  - No longer any need for duplicate pointers
  - No longer any need for explicit memory transfer
- Can naturally use more complicated data structures more naturally
  - Deep copies no longer needed for pointers to structures containing pointers
  - For example, you can construct a linked list on the host and traverse it on the device

Linked List without Unified Memory

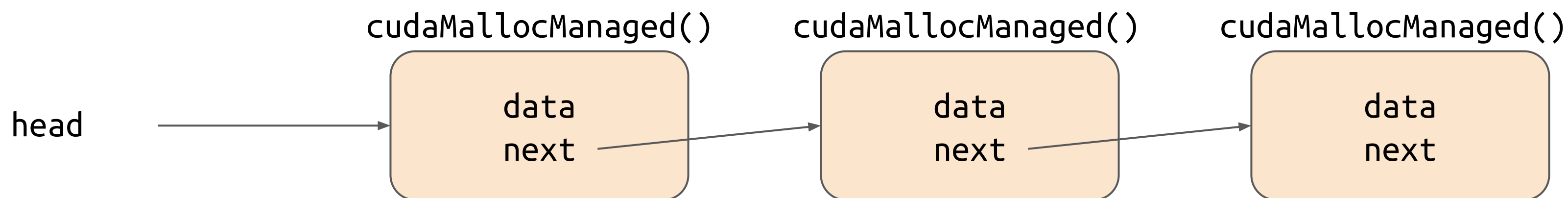


Linked List with Unified Memory

Nodes are allocated in unified memory

All pointers (head and all `next` pointers) are accessible on host and device

We can construct the list on the host, pass head to the device, and simply traverse



Unified memory

Performance of Unified Memory

- Can be nearly as fast as explicitly managed memory
- Until very recently, kernel launches caused all host-resident pages to be flushed to the GPU
- The Pascal architecture (2016) includes support for GPU page faults
  - A kernel can be launched with memory resident on the host
  - Page faults cause memory to be migrated to the GPU as needed

Conclusion / Takeaways

- cudaMemcpy requires data on the host to be in pinned memory
  - If the data is not already in pinned memory, the CUDA runtime will copy it to pinned memory and then to the device
- There are a number of options for transferring data from CPU to GPU with different ease of use and performance
- The CUDA CPU / GPU interface has been (and still is) evolving over time

Sources

https://www.wikipedia.org/

CUDA C Best Practices Guide.
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz4fVWqoMTH

Cheng, John, Max Grossman, and Ty McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.

Hwu, Wen-mei, and David Kirk. "Programming massively parallel processors." Special Edition 92 (2009).

Wilt, Nicholas. The cuda handbook: A comprehensive guide to gpu programming. Pearson Education, 2013.