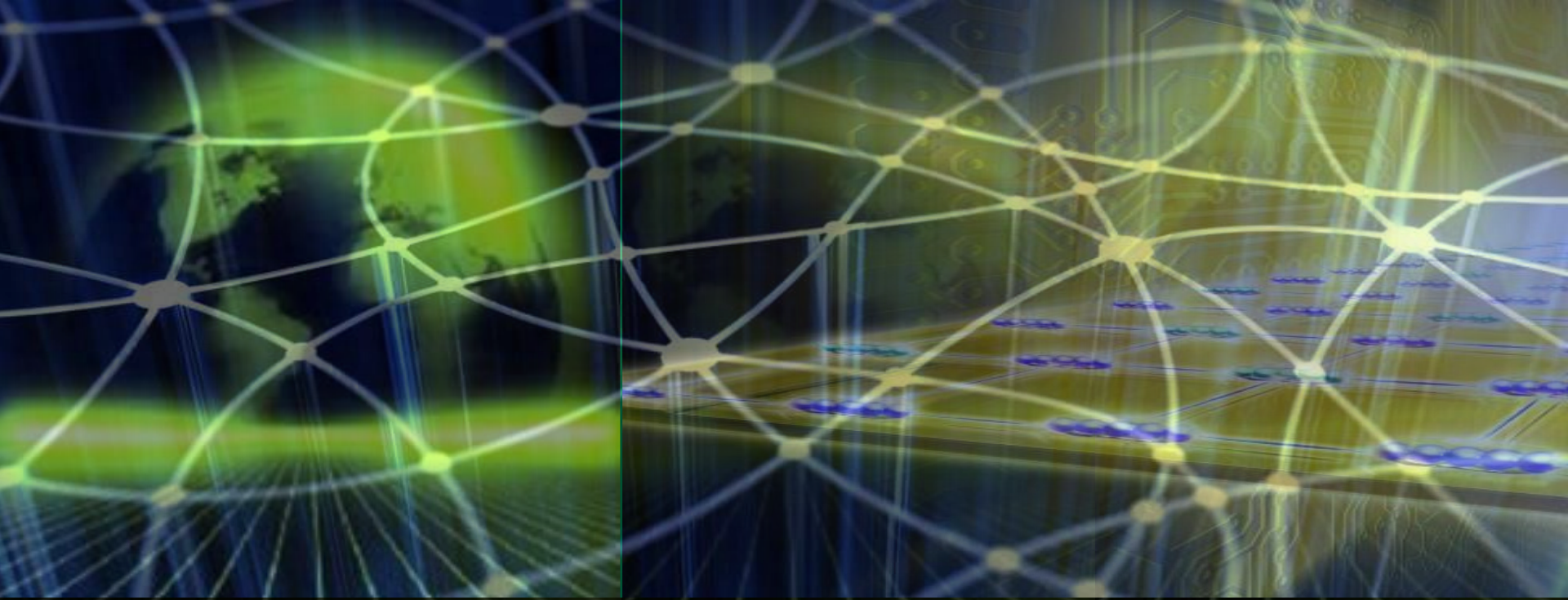


Memory Management

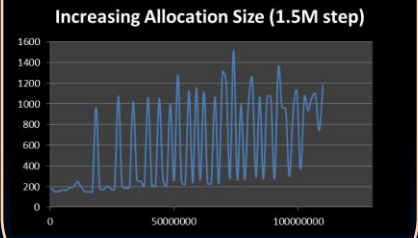
Tips, Tricks & Techniques

Stephen Jones, SpaceX, GTC 2015

Conclusion

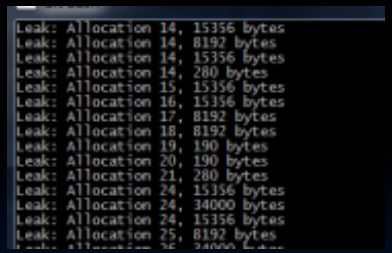
1. Wrap malloc/cudaMalloc with your own allocator

Non-Blockin,
High Performance
Sub-Allocation



Host/Device Data
Management

Leak Detection,
Debugging &
Profiling



Conclusion

2. There are three types of memory allocation

Persistent, Long-Lived Storage

For allocations spanning multiple program iterations
- main data storage
- C++ objects & configuration data

Working Space, Lifetime Of Single Iteration

For data which does not persist outside of one iteration
- per-iteration derived quantities
- operation working space, double buffers, etc.

Temporary, Local Allocation

For transient allocations with single-procedure lifetime
- local queues, stacks & objects
- function-scope working space

Take Control Of Memory Allocation

Take Control Of Memory Allocation

Debug & Leak Detection

Define your own allocate/free functions

Lightweight Allocators

Overload new & delete for all classes

Non-Blocking Allocation

Never call native malloc() or free()

Host/Device Management

It’s Easy!

C *malloc()* & *free()*

```
// Please don’t name these “malloc” & “free”
void *hostAlloc(size_t len) {
    return malloc(len);
}

void freeMem(void *ptr) {
    free(ptr);
}
```

C++ *new* & *delete*

```
// Every class should be “public AllocBase”
class AllocBase {
public:
    void *operator new(size_t len) {
        return hostAlloc(len);
    }

    void operator delete(void *ptr) {
        freeMem(ptr);
    }
};
```

Also Control Device Allocation

C *malloc()* & *free()*

```
// Please don't name these "malloc" & "free"
void *hostAlloc(size_t len) {
    return malloc(len);
}

void freeMem(void *ptr) {
    free(ptr);
}

void *deviceAlloc(size_t len) {
    void *ptr;
    cudaMalloc(&ptr, len);
    return ptr;
}
```

C++ *new* & *delete*

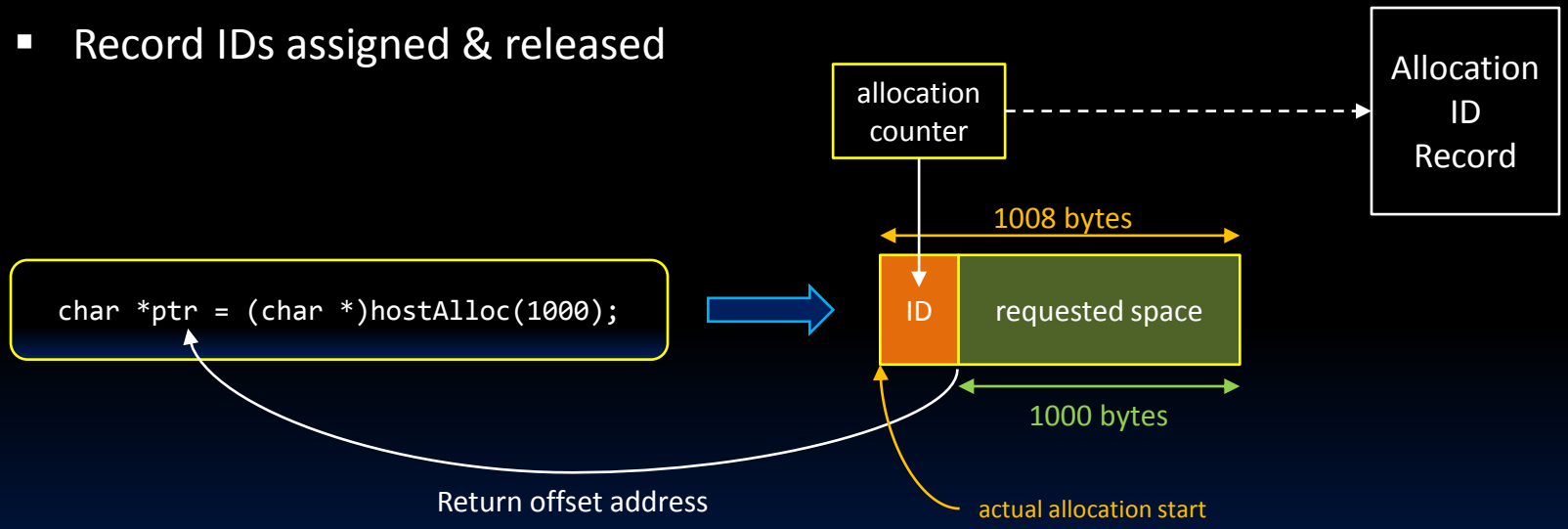
```
// Every class should be "public AllocBase"
class AllocBase {
public:
    void *operator new(size_t len) {
        return hostAlloc(len);
    }

    void operator delete(void *ptr) {
        freeMem(ptr);
    }
};
```

Allocation Tracking, Leak Detection & Profiling

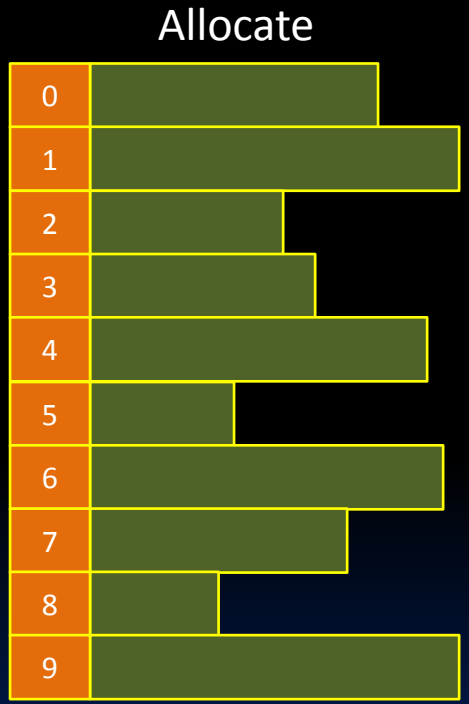
Memory Leak Detection

Track each allocation with unique identifier

- Allocate extra space for tracking ID
- Store ID in front of allocation
- Record IDs assigned & released



Memory Leak Detection



Memory Leak Detection

Memory Leak Detection

Identify Memory Leaks

Memory Leak Detection

```
// Use a C++11 atomic to count up allocation ownership
static std::atomic<long long>alloc_id = 0;
static std::vector<long long>allocationList;

void *hostAlloc(size_t len) {
    long long id = alloc_id++;    // Count up allocation ID
    allocationList[id] = 1;       // Record ID as “allocated”

    // Store allocation ID in front of returned memory
    void *ptr = malloc(len + 8);
    *(int *)ptr = id;
    return (char *)ptr + 8;
}

void freeMem(void *ptr) {
    // Extract allocation ID from front of allocation
    id = *(long long *)((char *)ptr – 8);
    allocationList[id] = 0;       // Record ID as “released”
    free((char *)ptr - 8);
}
```

Displaying Unreleased Allocations

For global-scope objects:

- Constructor called before *main()*
- Destructor called after *main()* exits

WARNING

- Order of static object construction & destruction is undefined
- Tracking objects should not interact

```
class TrackingObject {
public:
    // Set up initial data in constructor
    TrackingObject() {
        InitTrackingData();
    }

    // Analyse tracking data in destructor
    virtual ~TrackingObject() {
        ProcessTrackingData();
    }

    virtual void InitTrackingData() {}
    virtual void ProcessTrackingData() {}
};

// Create global-scope static object. Destructor
// is called automatically when program exits.
static TrackingObject dataTracker;
```

Displaying Unreleased Allocations

```
// Walks the allocation list looking for unallocated data
class AllocationTracker : public TrackingObject {
public:
    void ProcessTrackingData() {
        for( long long i=0; i<alloc_id; i++ ) {
            if( allocationList[i] != 0 ) {
                printf("Allocation %d not freed\n", i);
            }
        }
    }
}

// Creates a tracker which will be called on program shutdown
static AllocationTracker __allocationTracker;
```

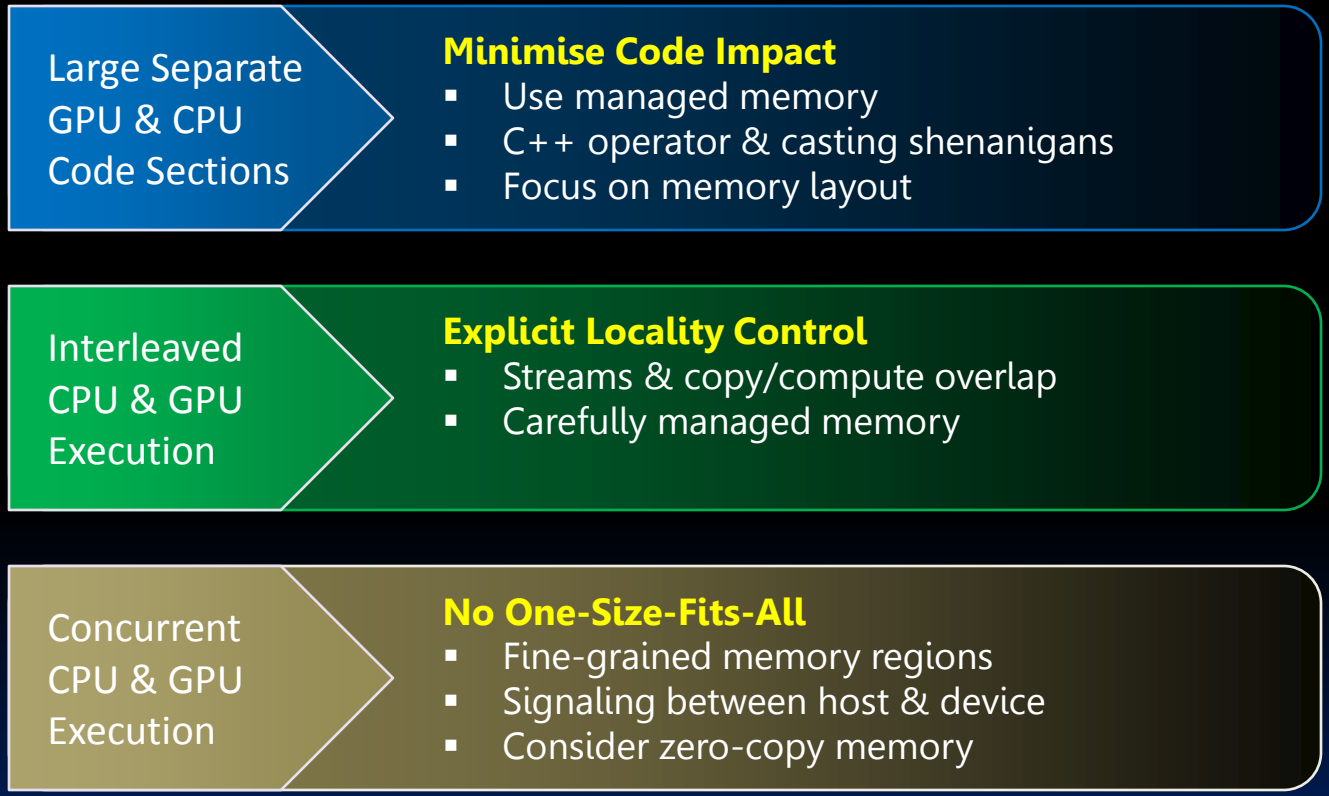
Complete Leak Tracking Code

```
// Auto display of memory leaks
static std::atomic<long long>alloc_id = 0;
static std::vector<long long>allocationList;

class AllocationTracker {
public:
   void ~AllocationTracker() {
      for( long long i=0; i<alloc_id; i++ ) {
          if( allocationList[i] != 0 ) {
             printf("Allocation %d not freed\n", i);
          }
      }
   }
}
static AllocationTracker __allocationTracker;
```

```
// Allocator with leak tracking
void *hostAlloc(size_t len) {
    long long id = alloc_id++;
    allocationList[id] = 1;

    void *ptr = malloc(len + 8);
    *ptr = id;
    return (char *)ptr + 8;
}

void freeMem(void *ptr) {
    id = *(long long *)((char *)ptr – 8);
    allocationList[id] = 0;
    free((char *)ptr - 8);
}
```

Host / Device Data Management

Managing Data Movement



Always Use Streams

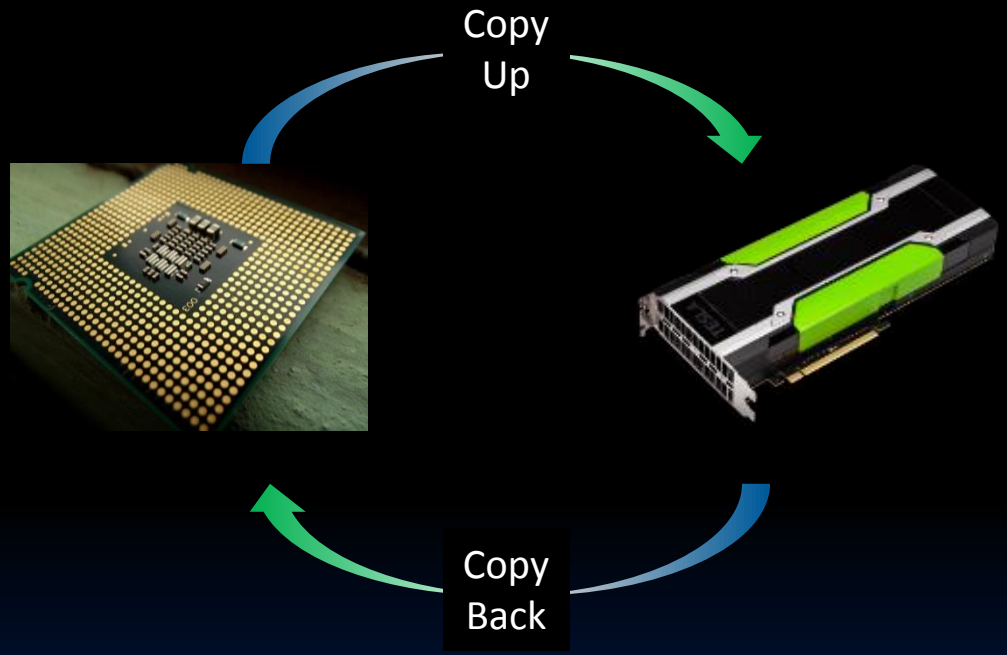
Always Use Streams

Whenever you launch a kernel

Whenever you copy data

Whenever you synchronize

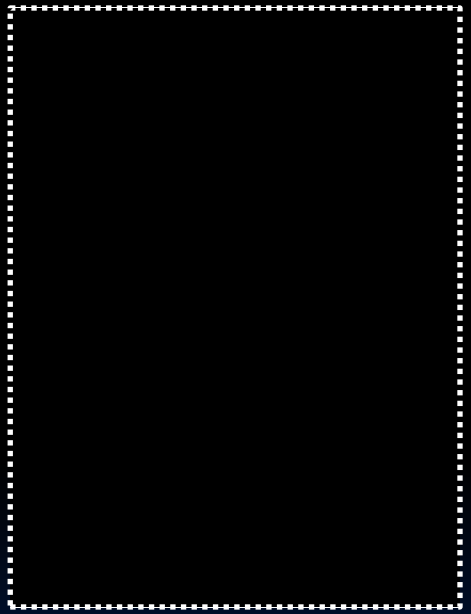
Streams & Copy/Compute Overlap



Tesla & Quadro GPUs support bi-directional copying

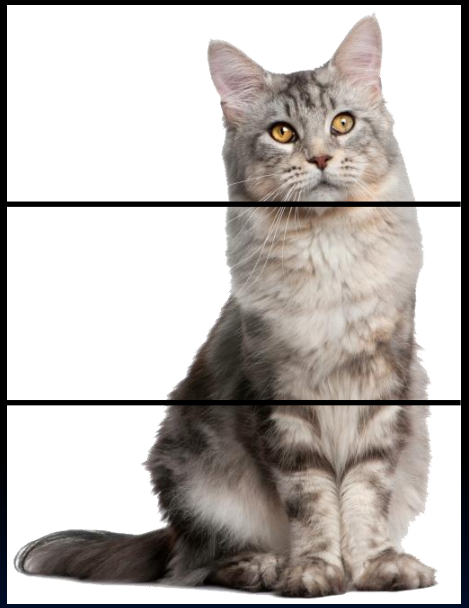
Streams & Copy/Compute Overlap

CPU

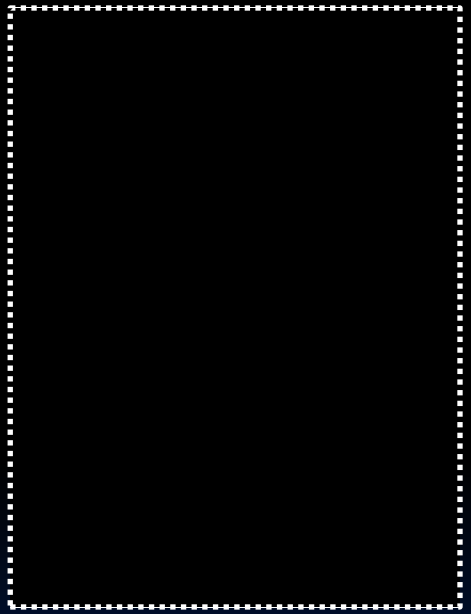


GPU

Streams & Copy/Compute Overlap

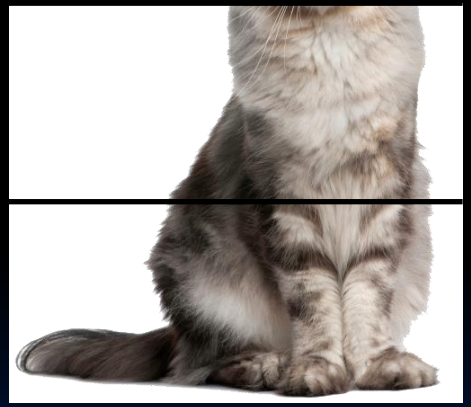


CPU

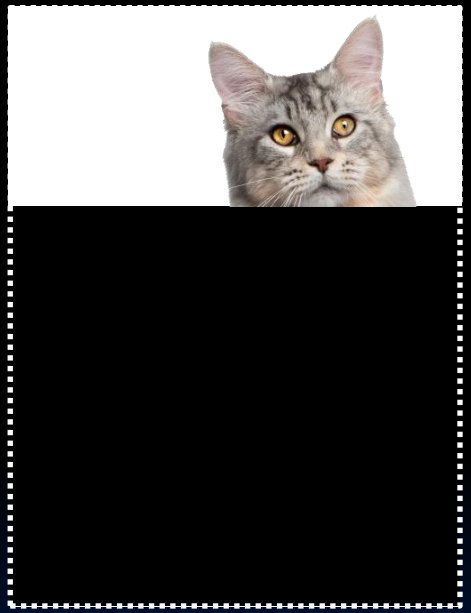


GPU

Streams & Copy/Compute Overlap



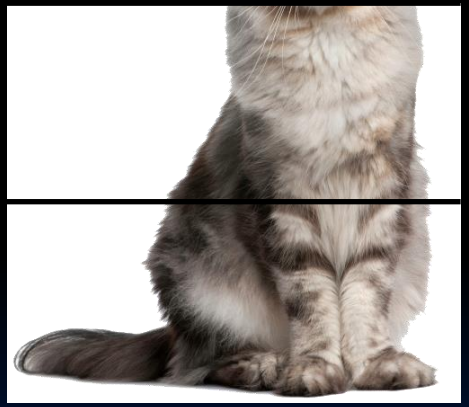
CPU



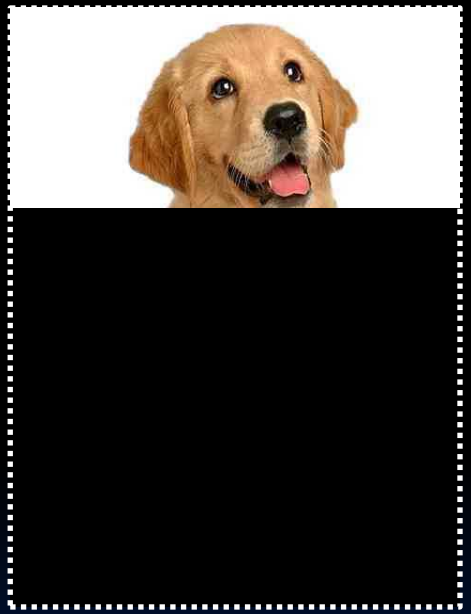
GPU

Step 1

Streams & Copy/Compute Overlap



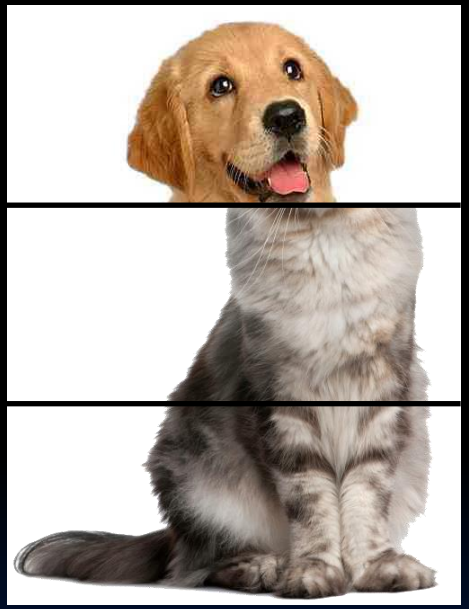
CPU



GPU

Step 2

Streams & Copy/Compute Overlap



CPU

GPU

Step 3

Streams & Copy/Compute Overlap

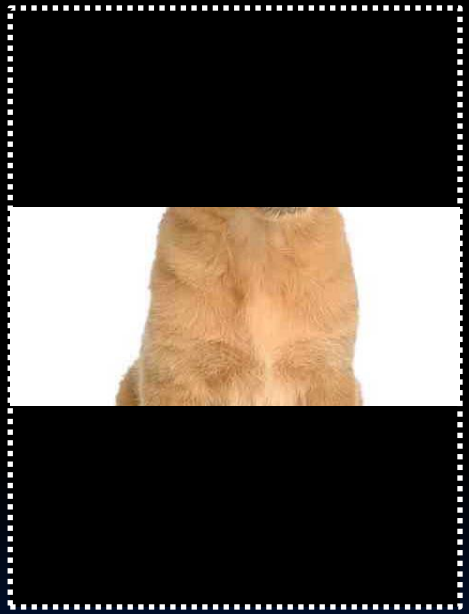
CPU

GPU

Step 4

Streams & Copy/Compute Overlap

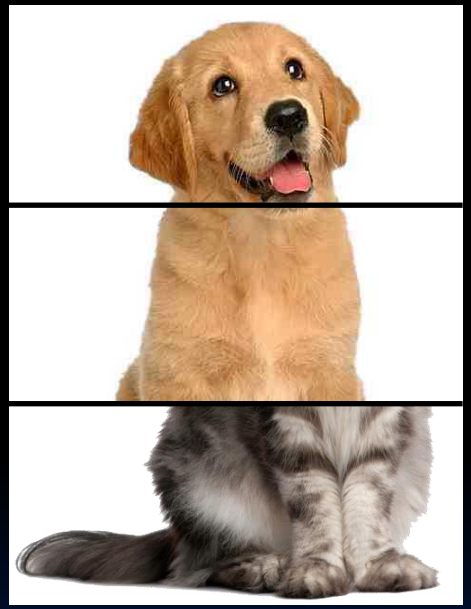
CPU



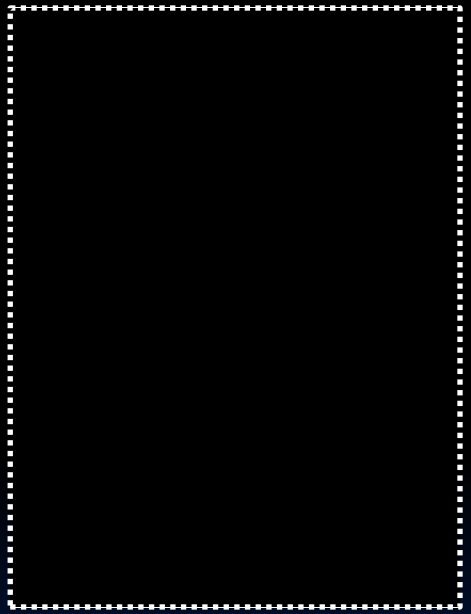
GPU

Step 5

Streams & Copy/Compute Overlap



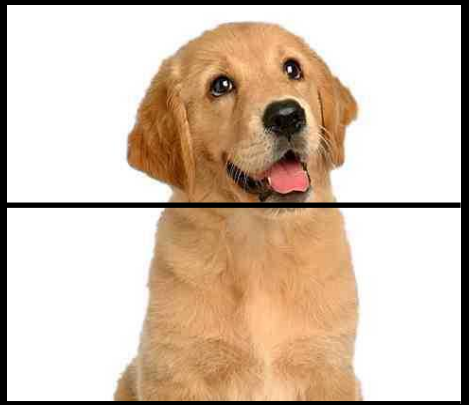
CPU

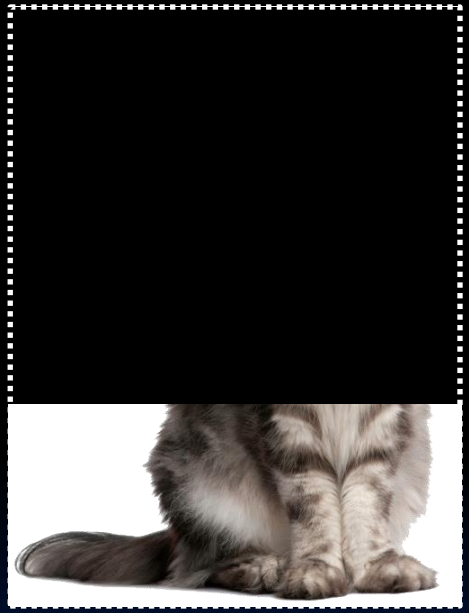


GPU

Step 6

Streams & Copy/Compute Overlap



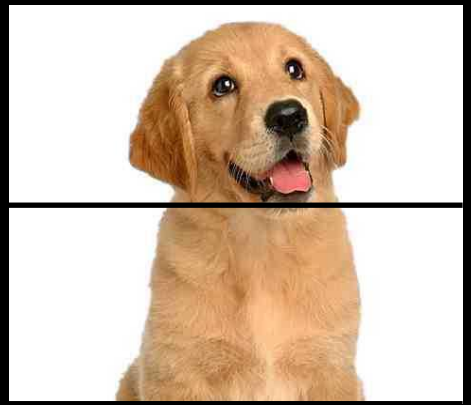


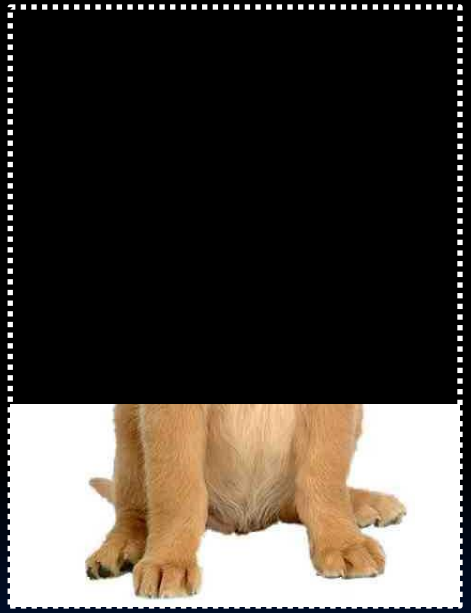
CPU

GPU

Step 7

Streams & Copy/Compute Overlap





CPU

GPU

Step 8

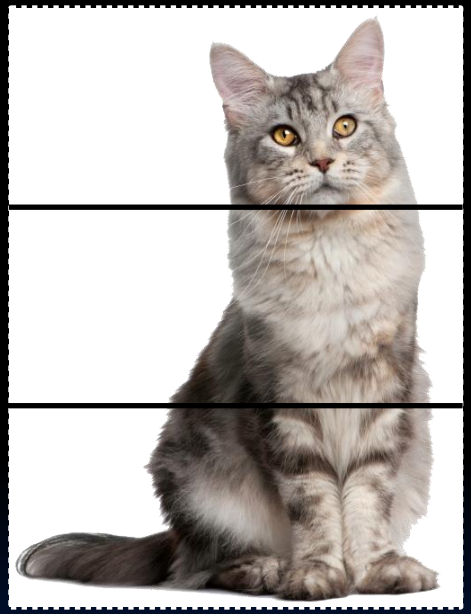
Streams & Copy/Compute Overlap

CPU

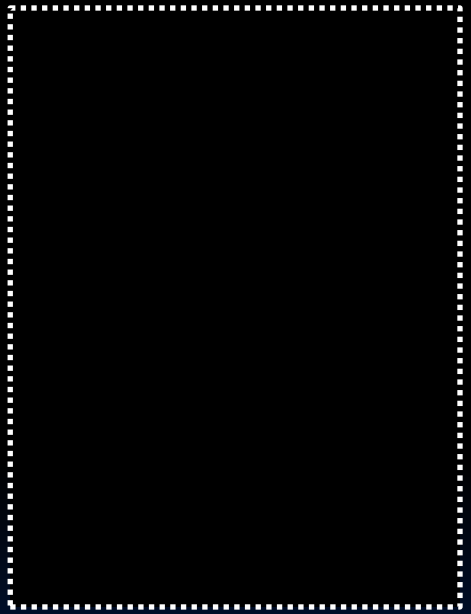
GPU

Step 9

Streams & Copy/Compute Overlap

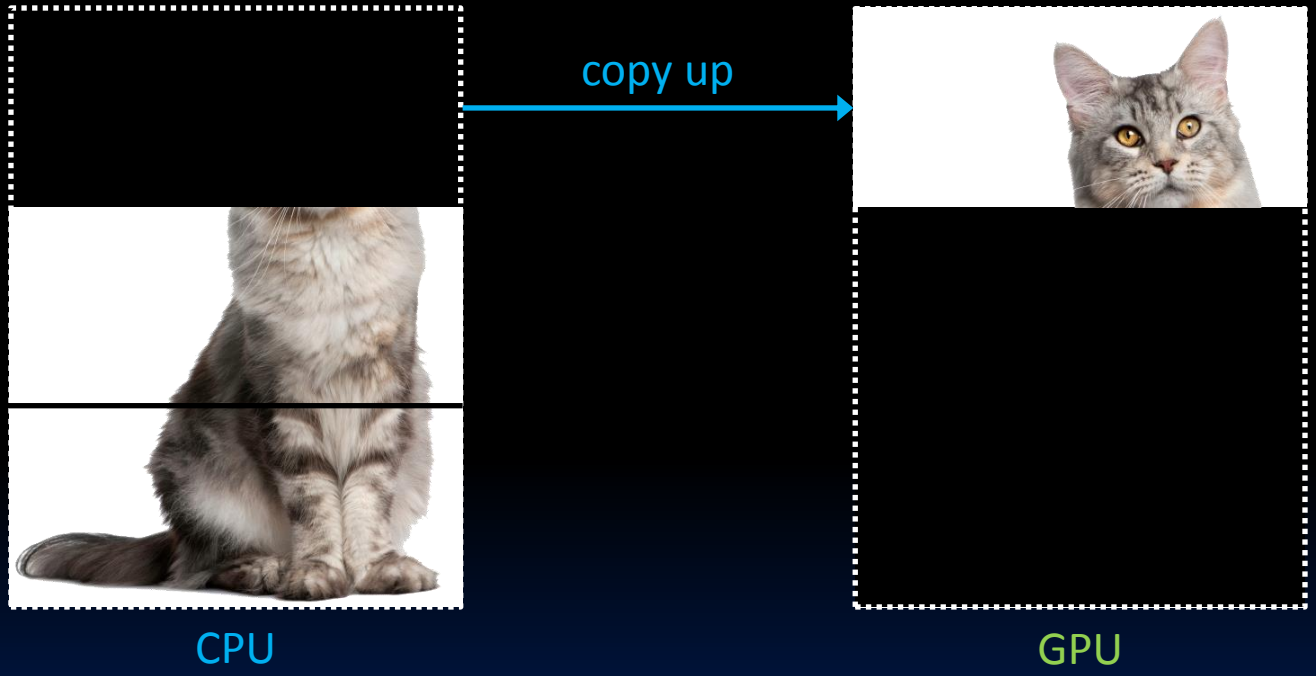


CPU



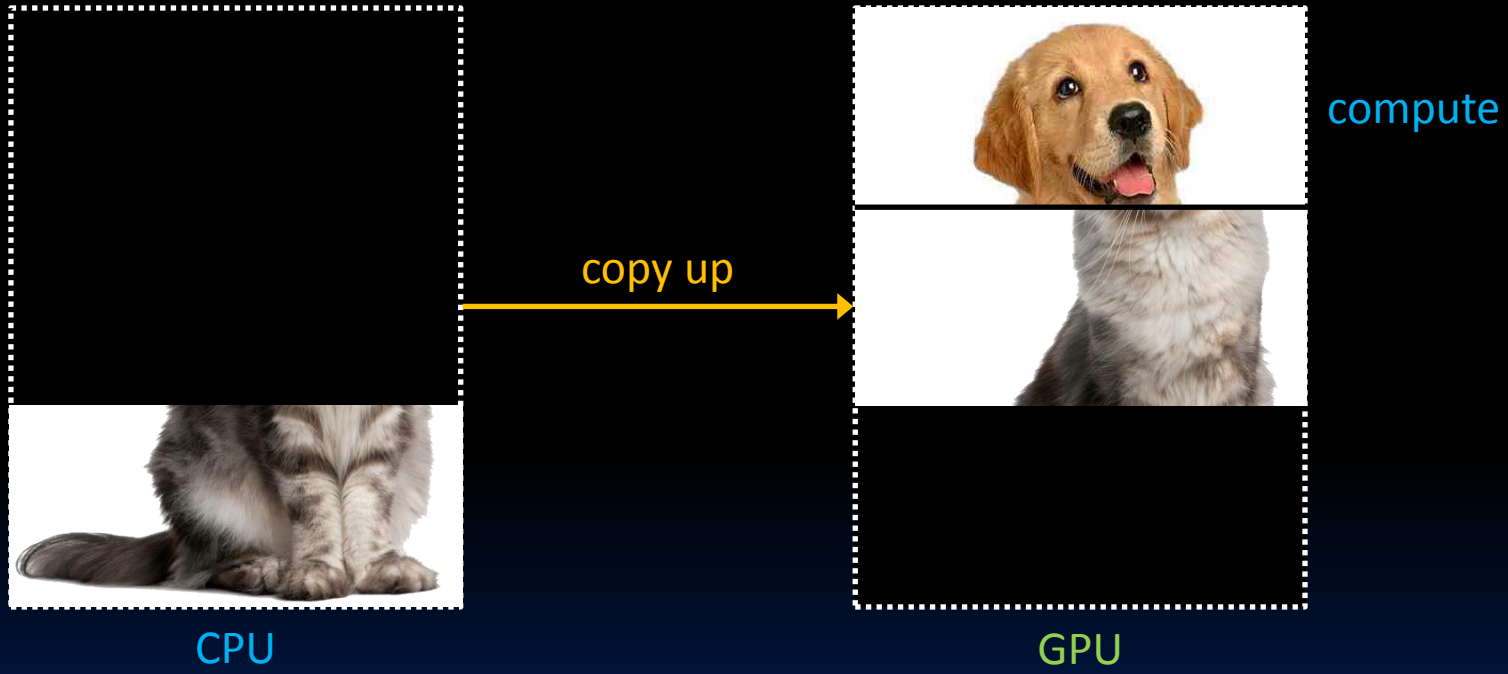
GPU

Streams & Copy/Compute Overlap



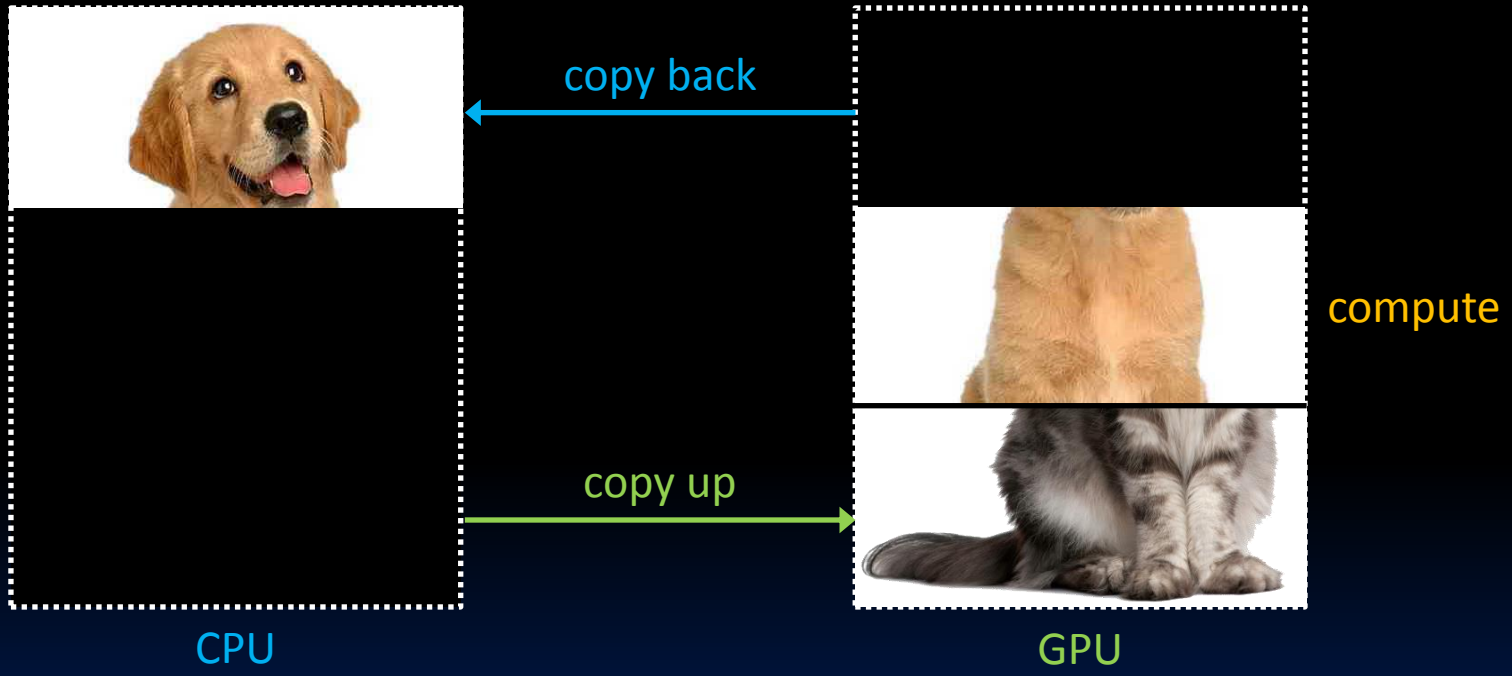
Step 1

Streams & Copy/Compute Overlap



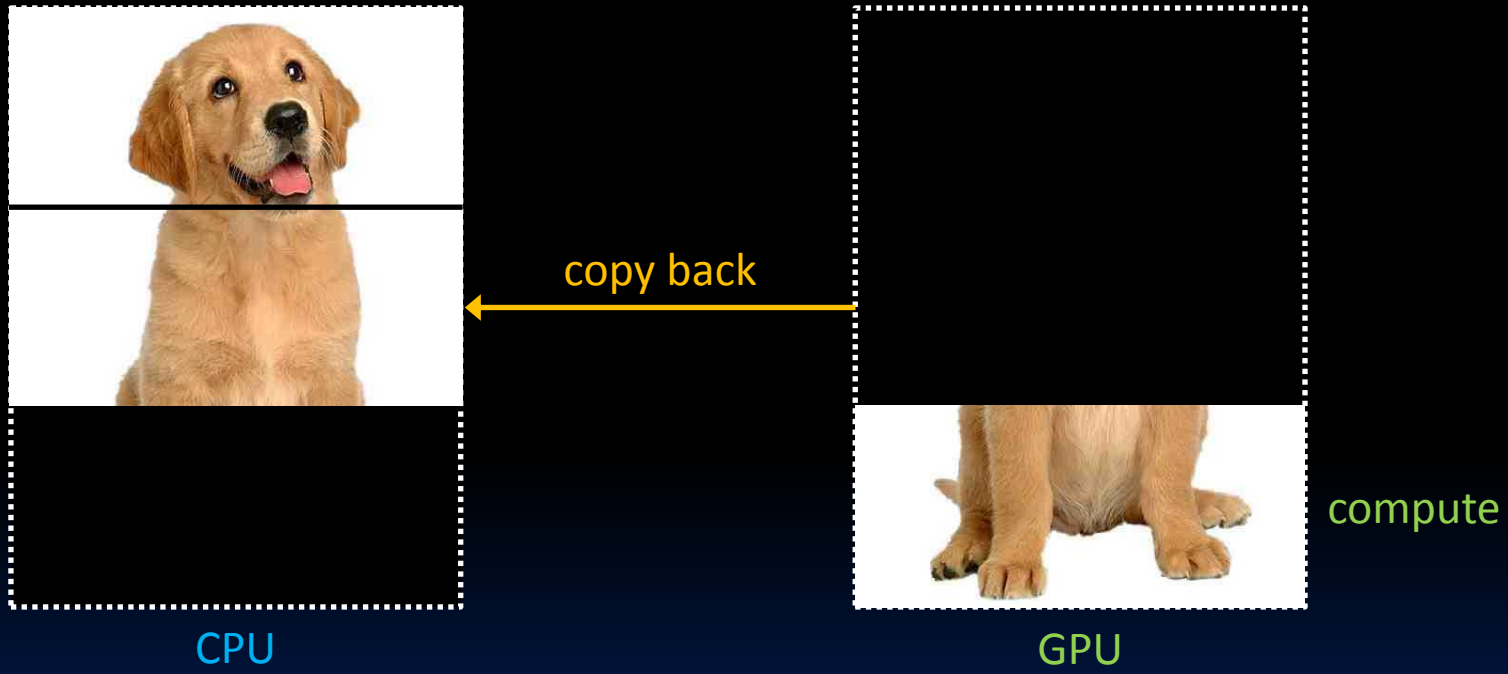
Step 2

Streams & Copy/Compute Overlap



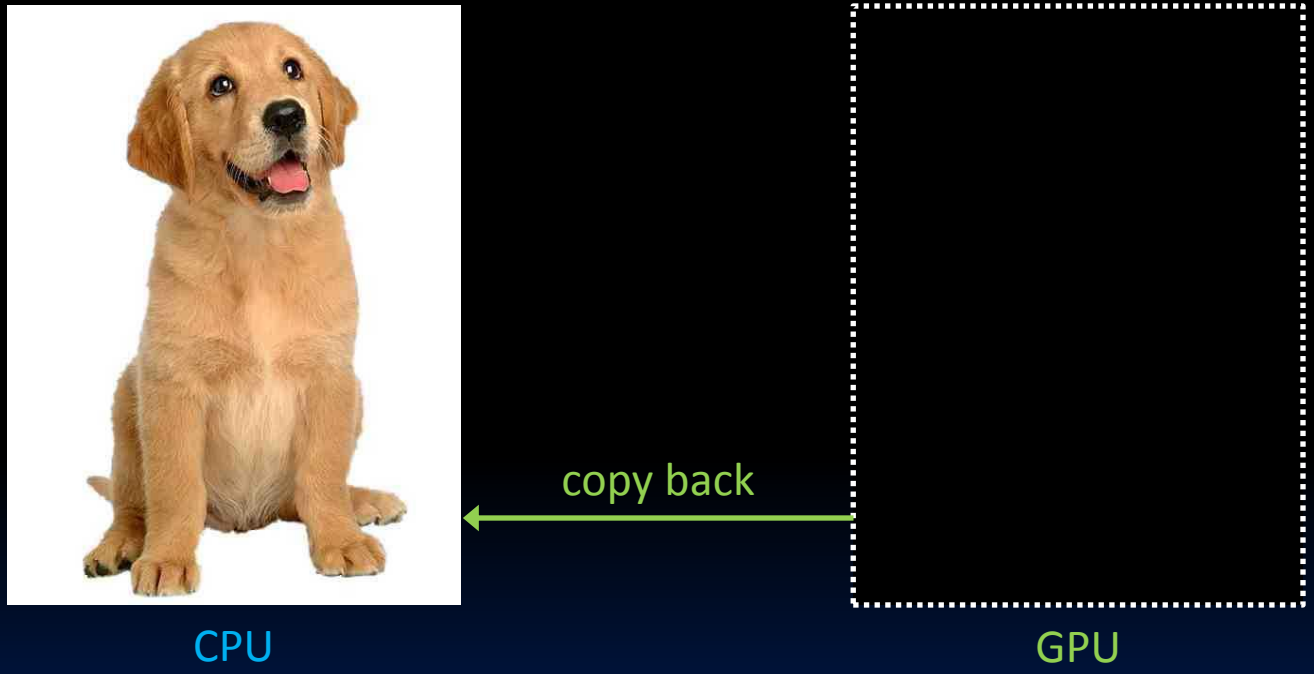
Step 3

Streams & Copy/Compute Overlap



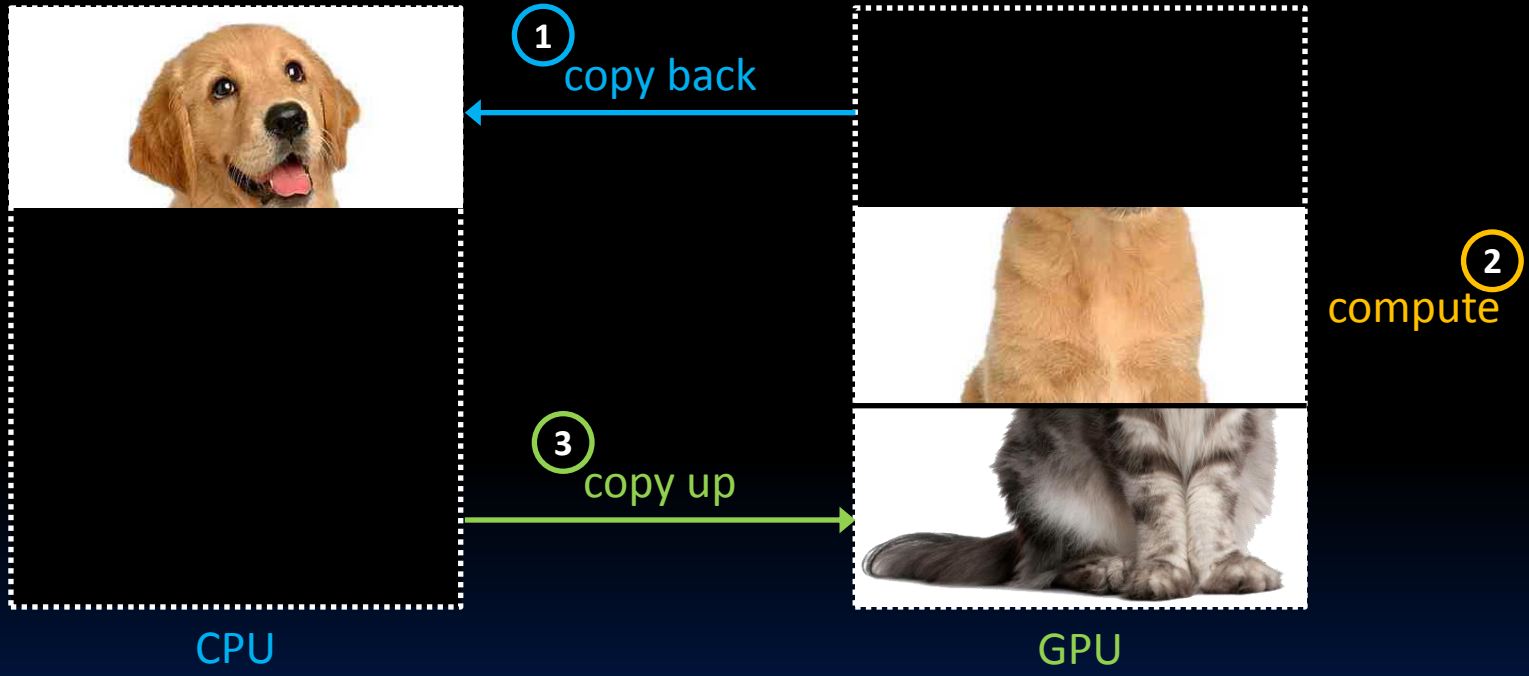
Step 4

Streams & Copy/Compute Overlap



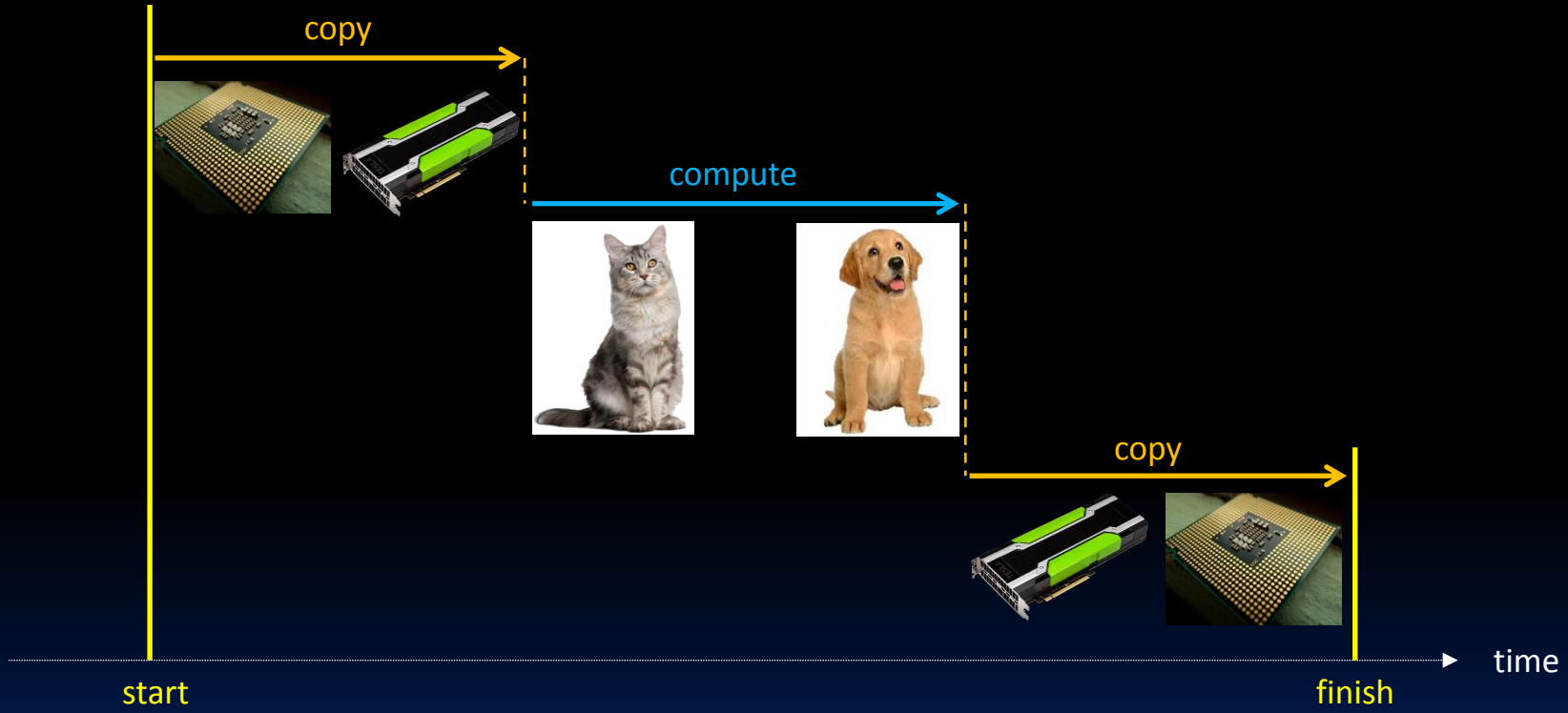
Step 5

Streams & Copy/Compute Overlap

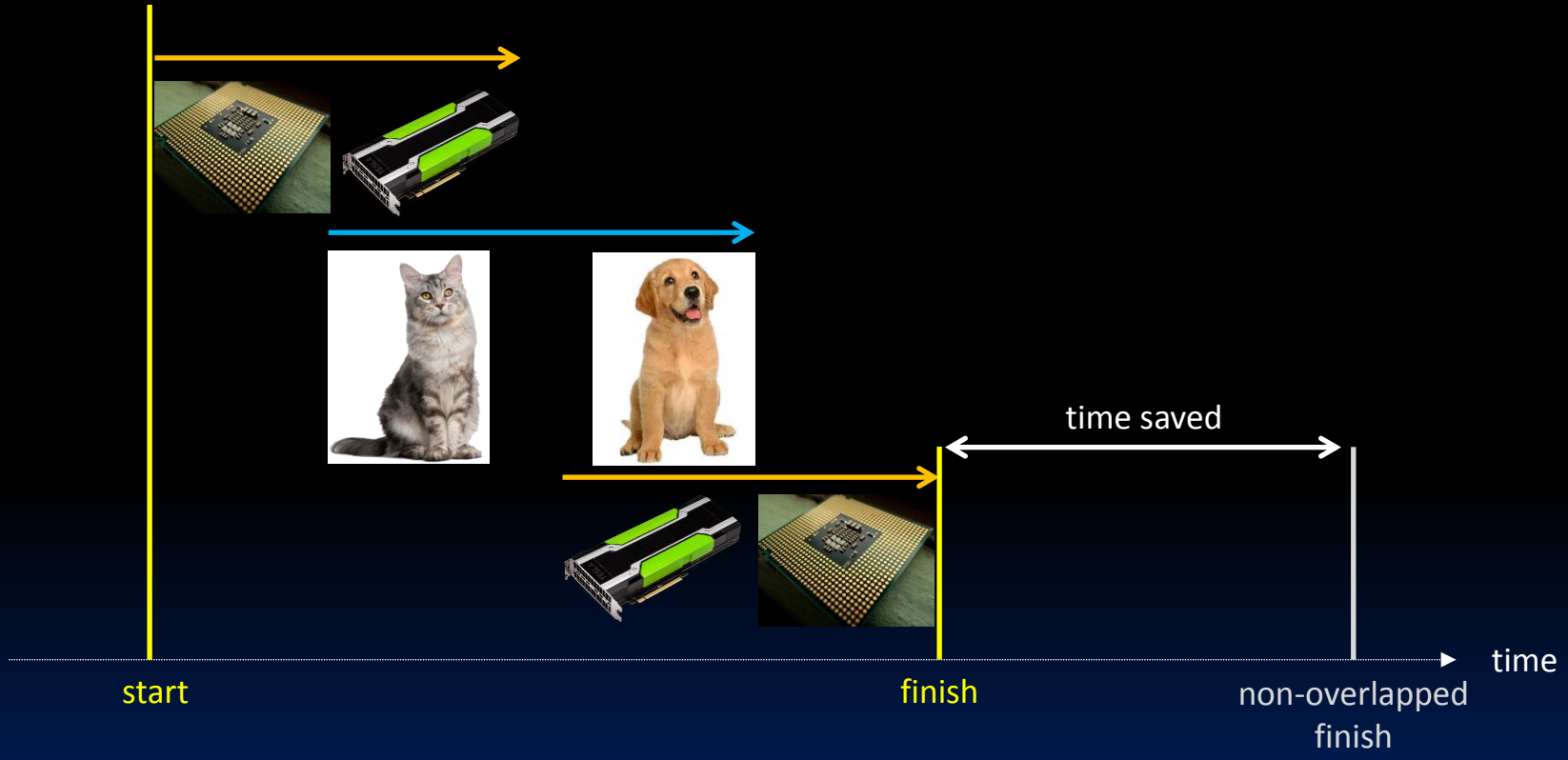


Three Simultaneous Operations

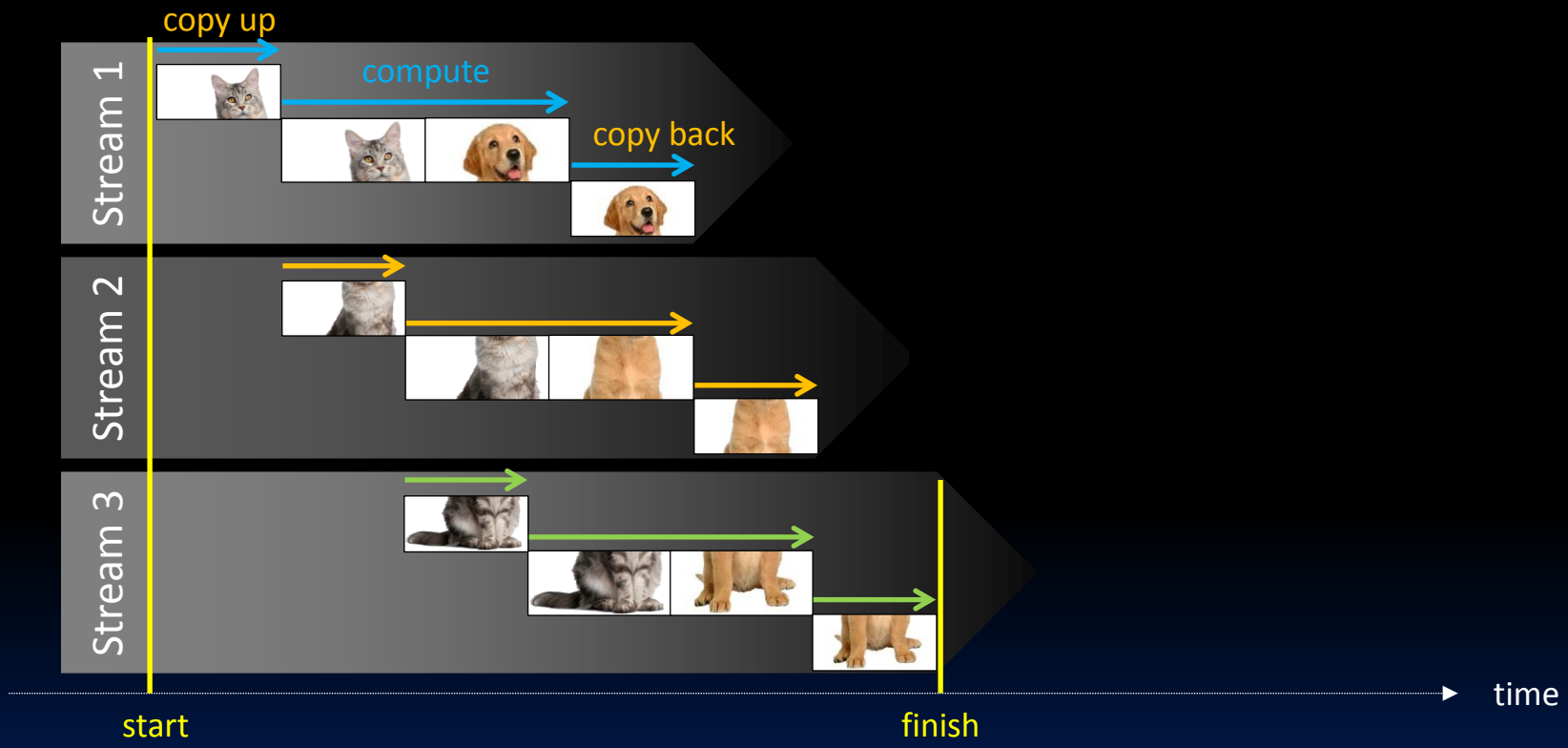
Overlapping Copy & Compute



Overlapping Copy & Compute



In More Detail...



Compute/Copy Overlap, in Code

```
// Convert cats to dogs in “N” chunks
void catsToDogs(char *cat, char *dog, int width, int height, int N) {
    // Loop copy+compute+copy for each chunk
    for( int h=0; h<height; h+=(height/N) ) {
        // Create a stream for this iteration
        cudaStream_t s;
        cudaStreamCreate( &s );

        // Allocate device data for this chunk
        char *deviceData;
        cudaMalloc( &deviceData, width * (height/N) );

        // Copy up then convert then copy back, in our stream
        cudaMemcpyAsync( deviceData, cat+h*width, ...hostToDevice, s );
        convert<<< width, height/N, 0, s >>>( deviceData );
        cudaMemcpyAsync( dog+h*width, deviceData, ...deviceToHost, s );

        // Free up this iteration’s resources
        cudaStreamDestroy( s );
        cudaFree( deviceData );
    }
}
```

Managed Memory

Very convenient for minimising code impact

- Can access same pointer from CPU & GPU, directly
- Data moves automatically
- Allows full-bandwidth access from GPU
- Tricky to use because of concurrency constraints (see next slides)

```
int *data;
cudaMallocManaged( &data, 10000000 );

data[100] = 1234;              // Access on CPU first
launch<<< 1, 1 >>>( data );    // Access on GPU second
```

Drawback Of Managed Memory

CPU cannot touch managed memory while the GPU is active

- "active" means any launch or copy since last synchronize()

```
int *data;
cudaMallocManaged( &data, 10000000 );

launch<<< 1, 1 >>>( data );   // Access on GPU first
data[100] = 1234;             // CPU access fails
                              // because GPU is busy
```

Drawback Of Managed Memory

CPU cannot touch managed memory while the GPU is active

- "active" means any launch or copy since last synchronize()
- Even if the GPU kernel is not actually using the data

```
int *data;
cudaMallocManaged( &data, 10000000 );

launch<<< 1, 1 >>>();          // GPU does not touch data
data[100] = 1234;              // CPU access still fails
                               // because GPU is busy!
```

“Attaching” Managed Memory

“Attach” reduces constraint to *’while a specific stream is active*’

- Allows CPU to touch some data while GPU is busy with other data

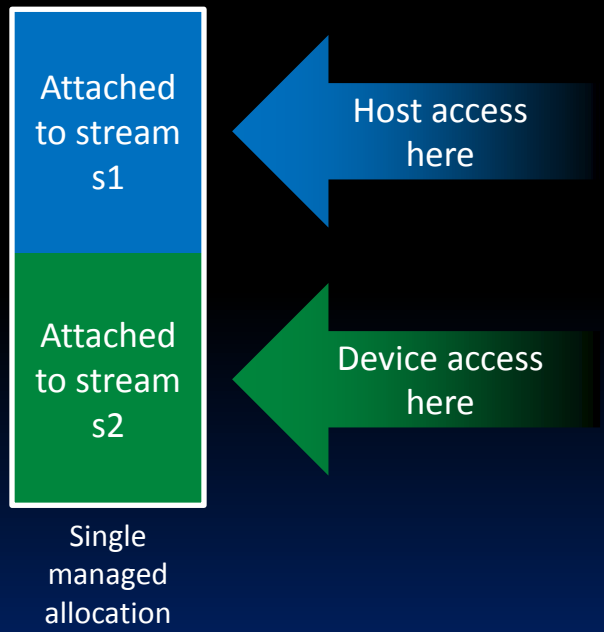
```
// Assume streams “s1” & “s2” exist
int *data;
cudaMallocManaged( &data, 10000000 );

// Associate “data” with stream s1
cudaStreamAttachMemAsync( s1, data );

// Launch GPU work on stream s2
launch<<< 1, 1, 0, s2 >>>();
data[100] = 1234;              // Access on CPU succeeds
```

Managed Memory Attach Tricks

Trick: You can attach just a part of an allocation

- Allows heterogeneous access to different parts of the same allocation

```
// Assume streams “s1” & “s2” exist
int *data;
cudaMallocManaged( &data, 65536);

// Associate half of “data” with stream s1
// and half with stream s2
cudaStreamAttachMemAsync( s1, data, 32768 );
cudaStreamAttachMemAsync( s2, data+32768, 32768 );

// Launch on stream s2 is fine, but you are
// responsible for not touching top half of data
launch<<< 1, 1, 0, s2 >>>( data );
data[100] = 1234;      // Access on CPU succeeds
```



Managed Memory Attach Tricks

Dirty Trick: Attach memory not used by a kernel to the CPU

- You can tell CUDA that you know best, and CPU-access is safe
- Must re-attach to a stream to use it on the device
- WARNING: Memory will not be shared with GPU while host-attached

```
// Assume stream s1 exists
int *data;
cudaMallocManaged( &data, 10000000 );

// Associate data with the CPU
cudaStreamAttachMemAsync( s1, data, 0, cudaMemAttachHost );

// Launch GPU work that doesn't use "data"
launch<<< 1, 1, 0, s1 >>>();
data[100] = 1234;                // Access on CPU succeeds
```

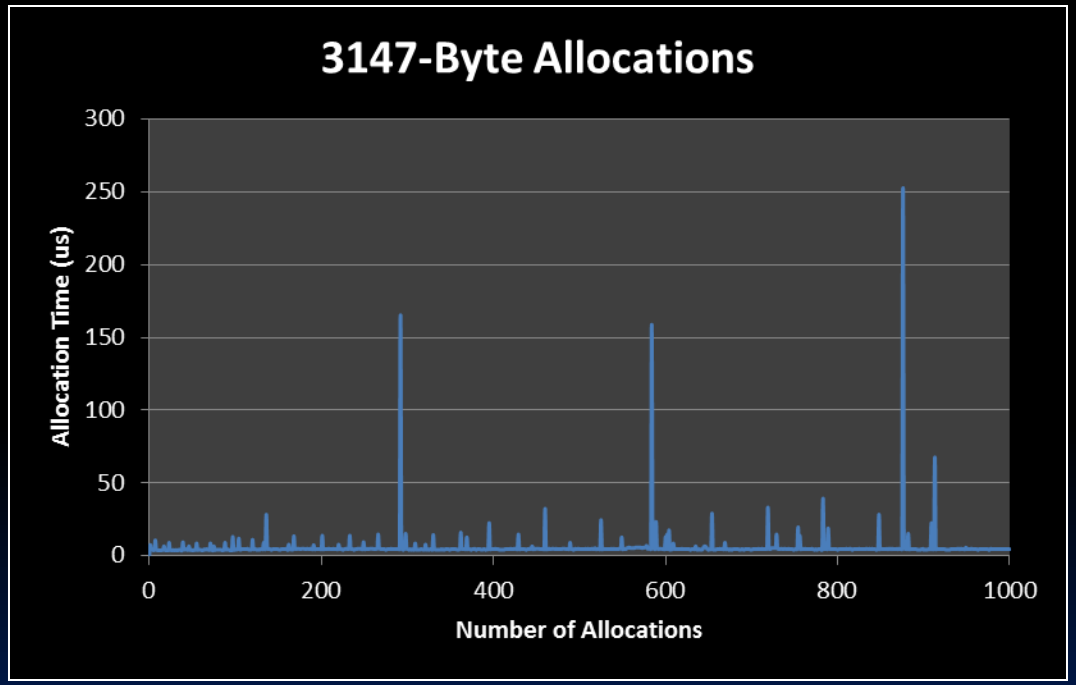
Owning GPU Memory Allocation

CUDA Memory Allocation Issues

GPU memory allocation can and does synchronize all streams

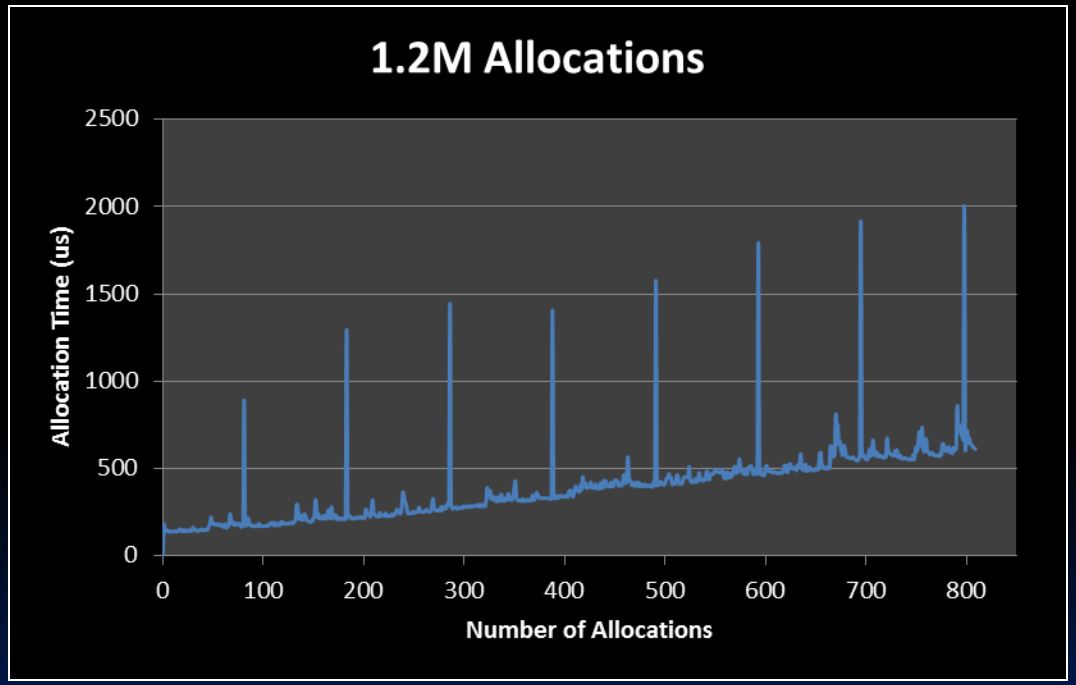
cudaMalloc() Behaviour Varies Widely

Repeated allocation of fixed size



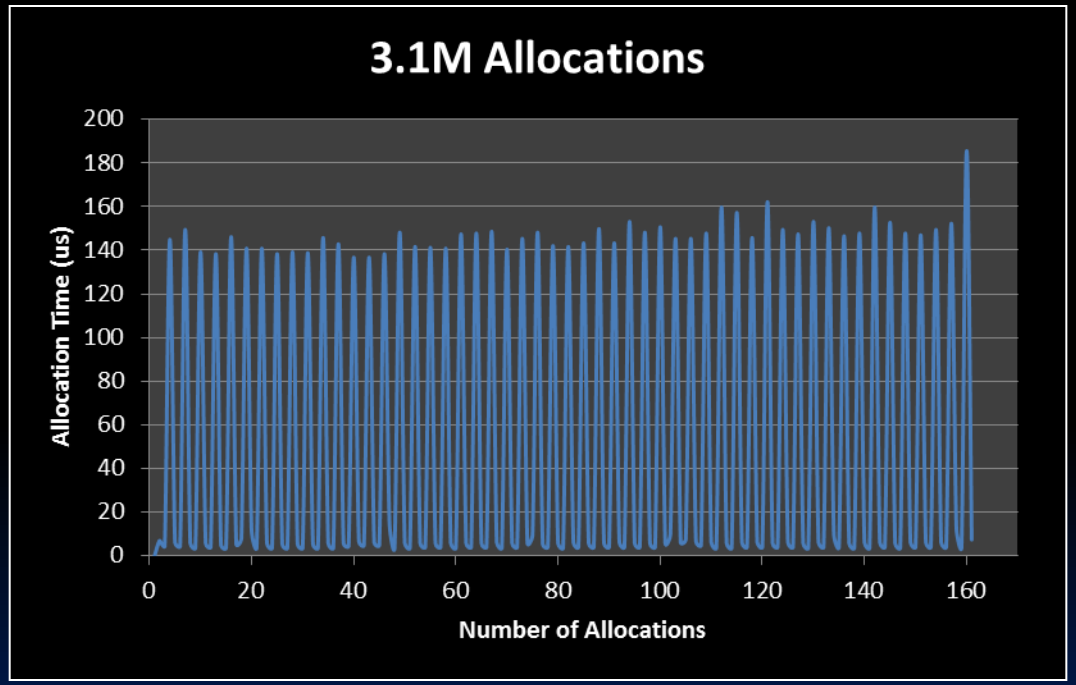
cudaMalloc() Behaviour Varies Widely

Repeated allocation of fixed size



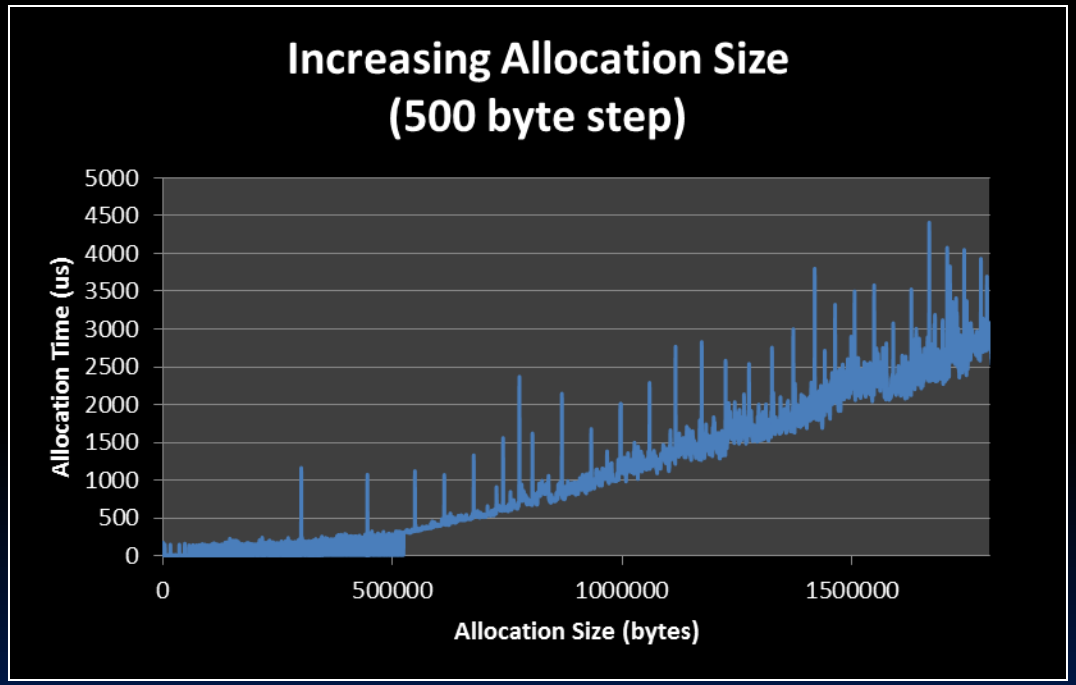
cudaMalloc() Behaviour Varies Widely

Repeated allocation of fixed size



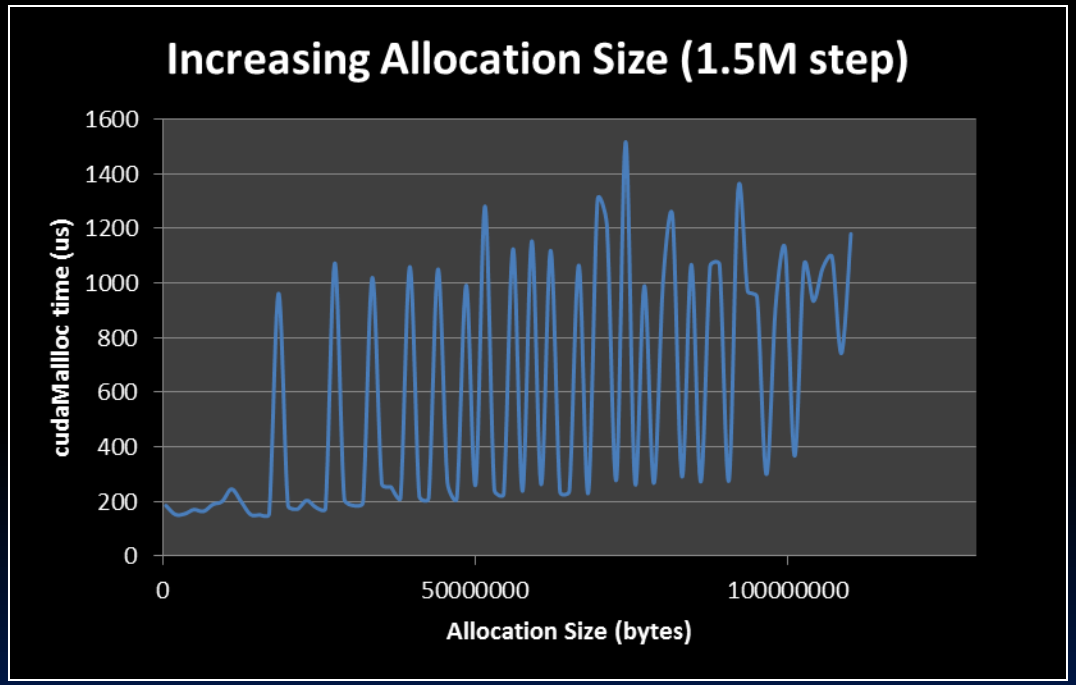
cudaMalloc() Behaviour Varies Widely

Repeated allocations of increasing size



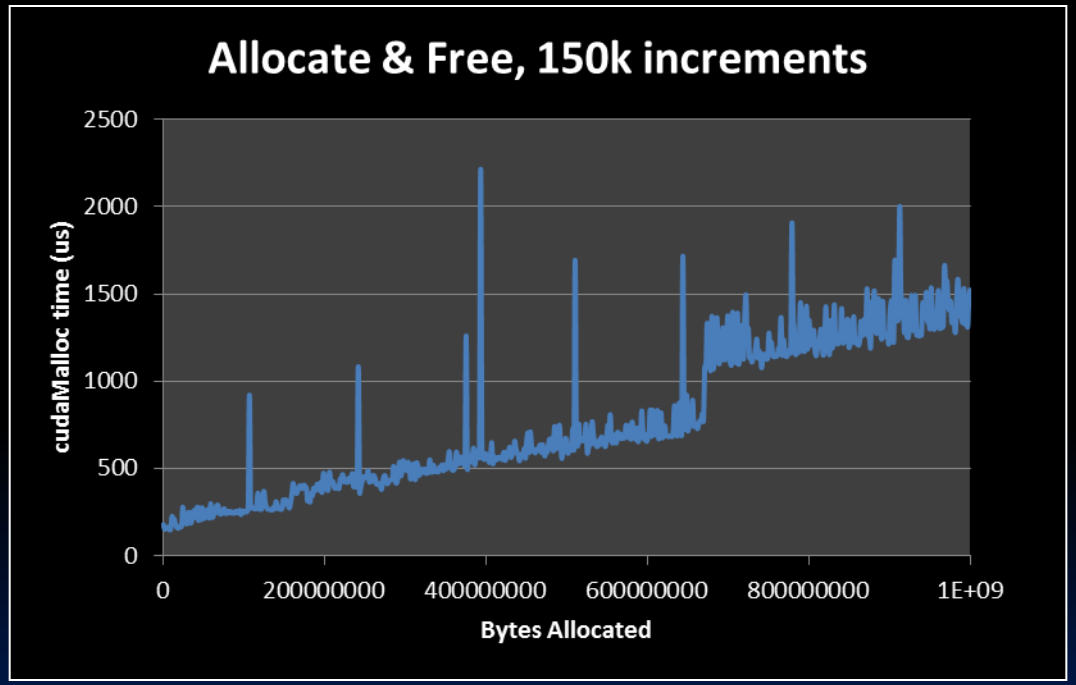
cudaMalloc() Behaviour Varies Widely

Repeated allocations of increasing size



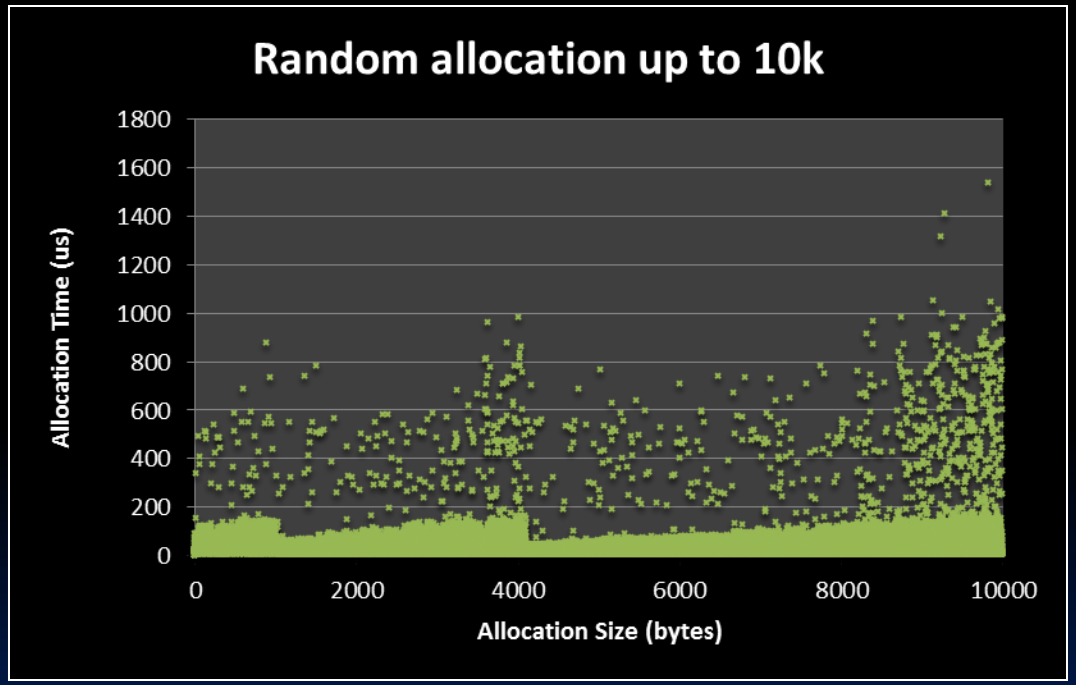
cudaMalloc() Behaviour Varies Widely

Mixed allocation & free, increasing size



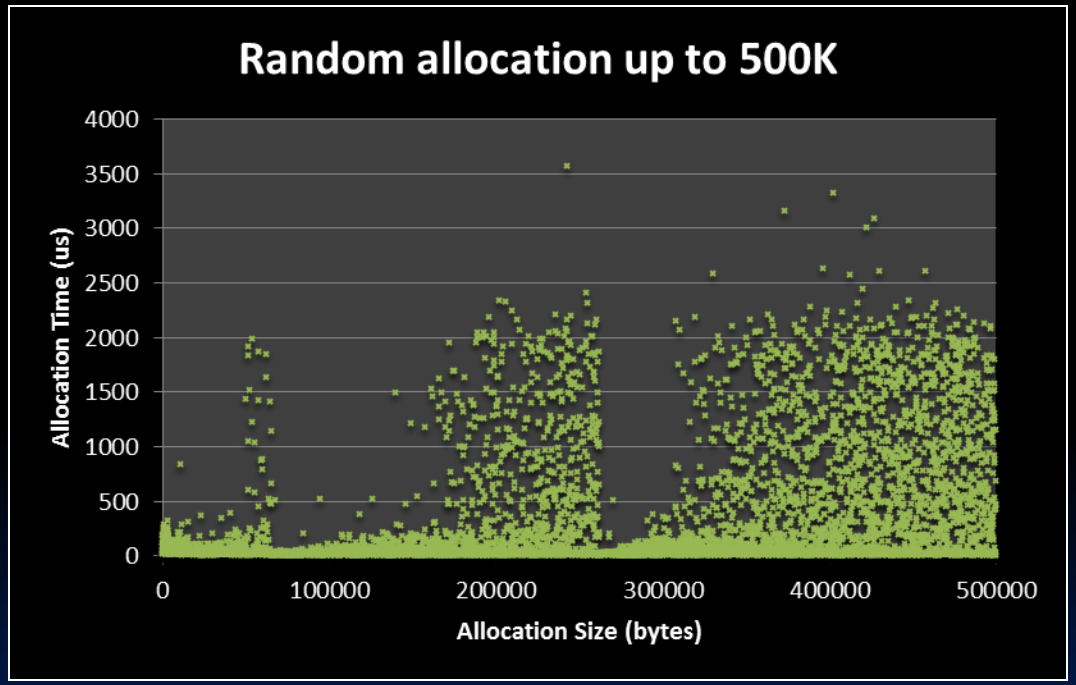
cudaMalloc() Behaviour Varies Widely

Time variation when allocating small blocks of data



cudaMalloc() Behaviour Varies Widely

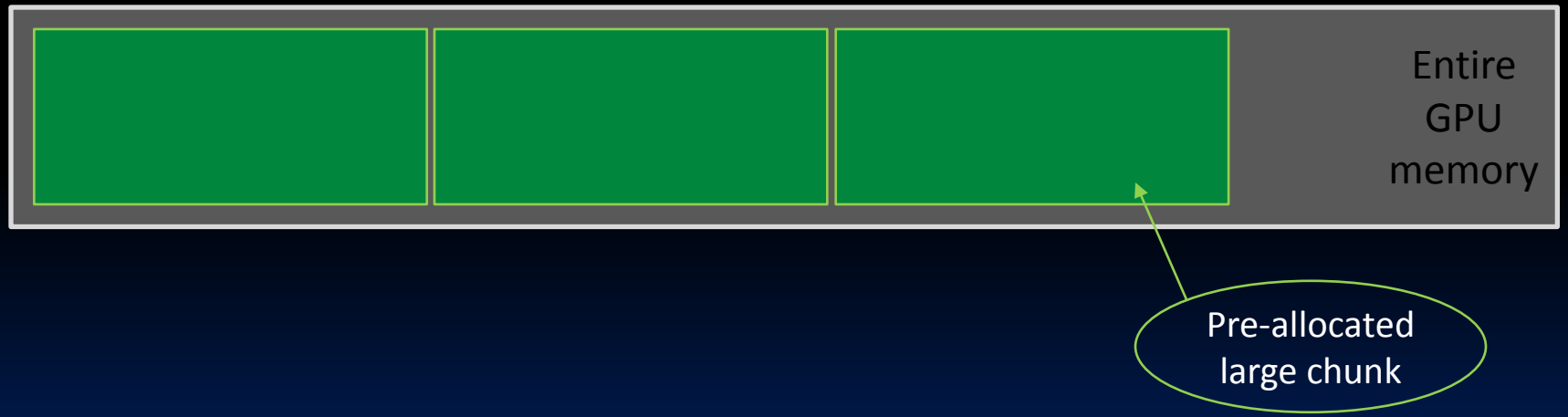
Time variation when allocating larger blocks of data



Roll-Your-Own Allocators

Sub-Allocators

1. Pre-allocate a one or more large chunks of memory

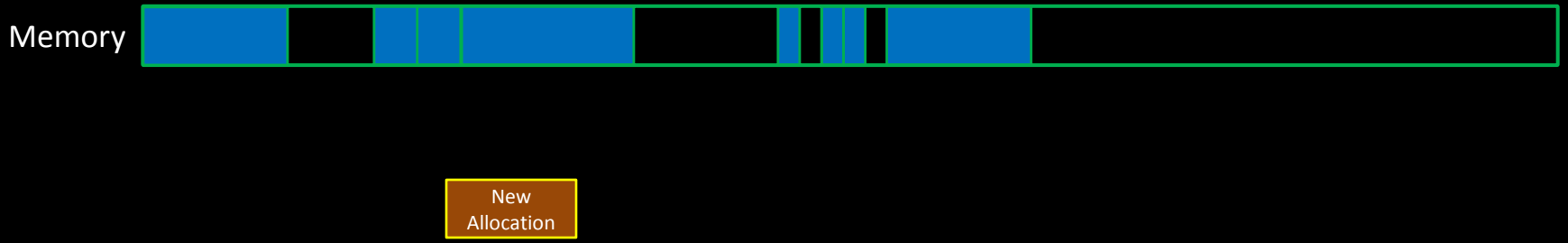


Sub-Allocators

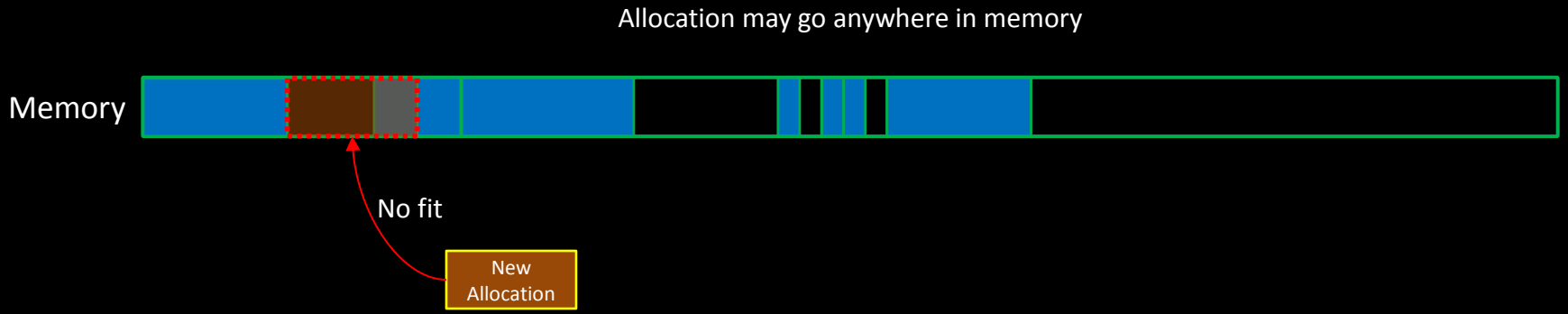
1. Pre-allocate a one or more large chunks of memory
2. Allocation requests then carve out pieces without having to touch the hardware

Heap Allocators

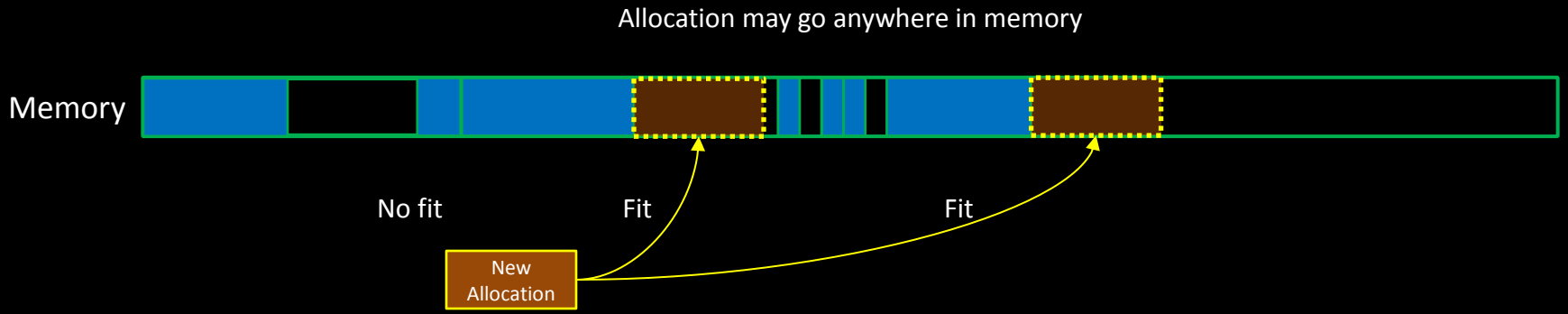
Allocation may go anywhere in memory



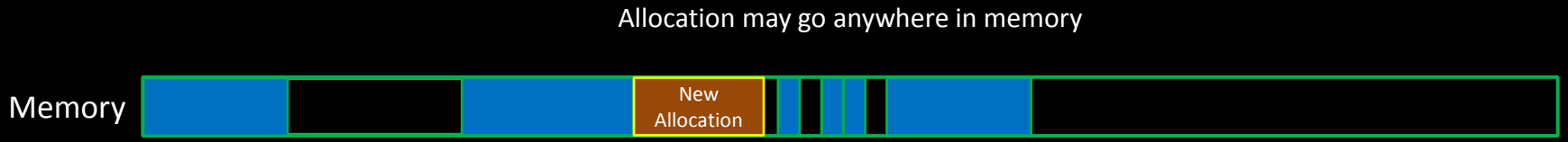
Heap Allocators



Heap Allocators

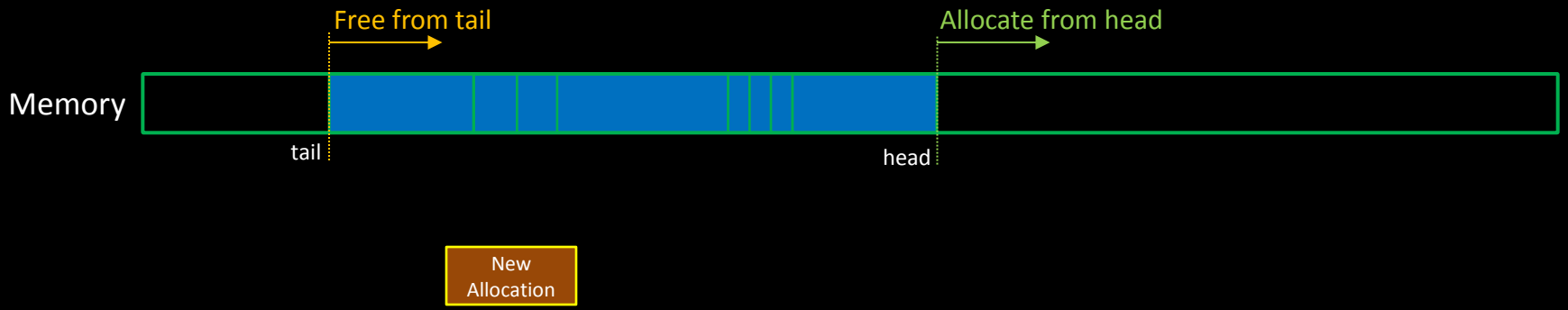


Heap Allocators

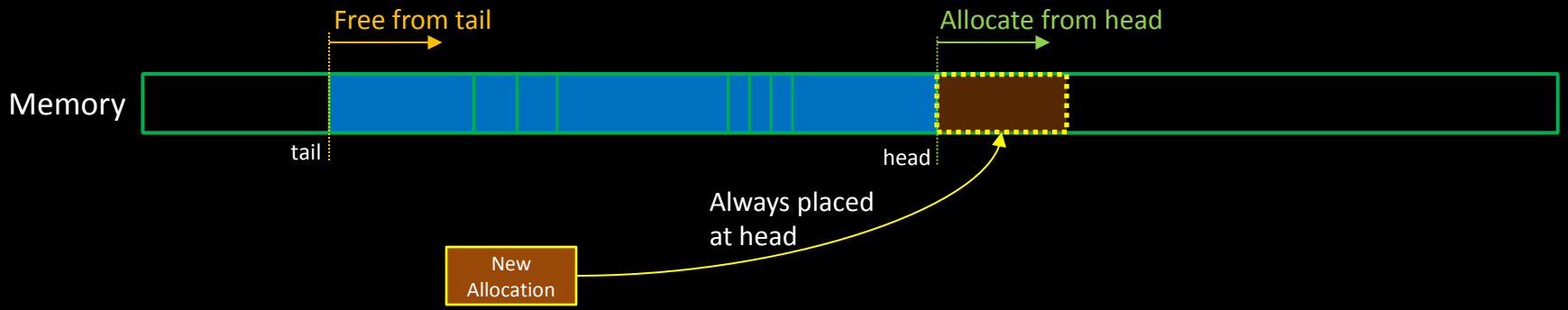


- New allocation must find next and/or best-fit free space in memory
- Free releases block in-place (fragmentation)
- Complex; countless approaches: SLAB, SLUB, red-black trees, etc.

Ring-Buffer Allocators



Ring-Buffer Allocators



Ring-Buffer Allocators

- New allocations always adds to head of buffer
- Free only permitted from tail – (out-of-order free = fragmentation)
- Fast, fairly simple, but long-lived allocations will block allocator

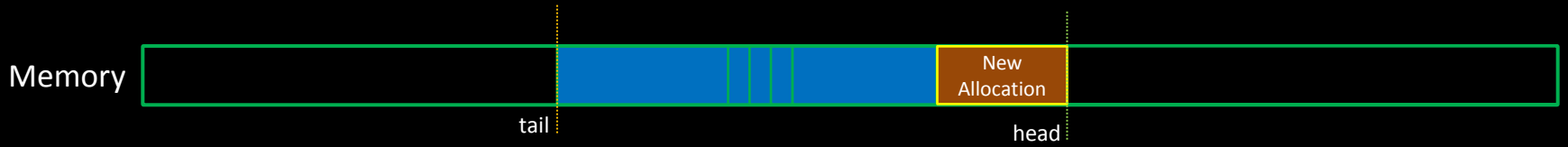
Ring-Buffer Allocators

- New allocations always adds to head of buffer
- Free only permitted from tail – (out-of-order free = fragmentation)
- Fast, fairly simple, but long-lived allocations will block allocator

Ring-Buffer Allocators

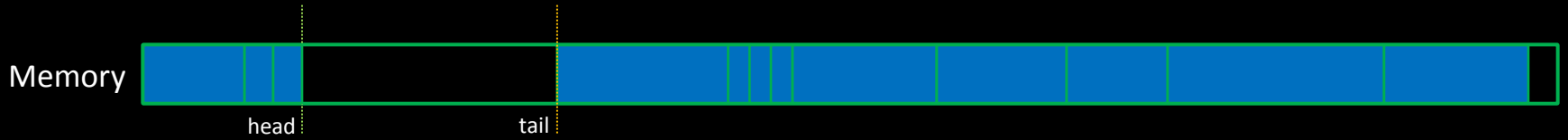
- New allocations always adds to head of buffer
- Free only permitted from tail – (out-of-order free = fragmentation)
- Fast, fairly simple, but long-lived allocations will block allocator

Ring-Buffer Allocators



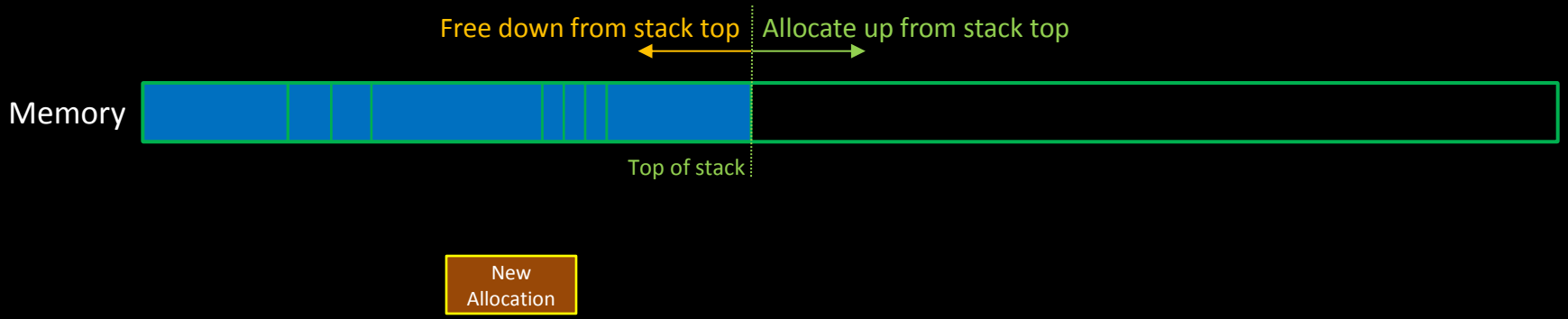
- New allocations always adds to head of buffer
- Free only permitted from tail – (out-of-order free = fragmentation)
- Fast, fairly simple, but long-lived allocations will block allocator

Ring-Buffer Allocators

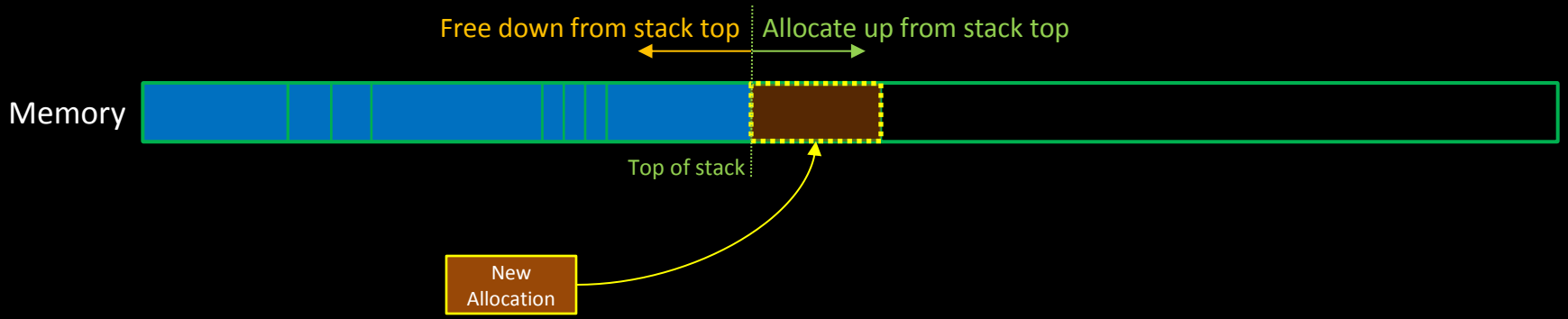


- New allocations always adds to head of buffer - wraps around at end
- Free only permitted from tail – (out-of-order free = fragmentation)
- Fast, fairly simple, but long-lived allocations will block allocator

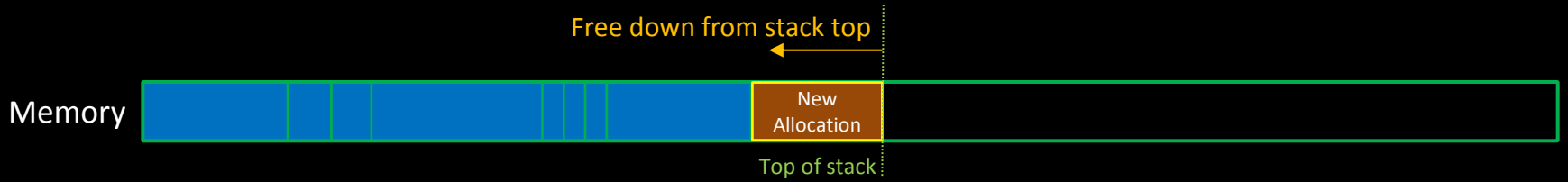
Stack Allocators



Stack Allocators



Stack Allocators

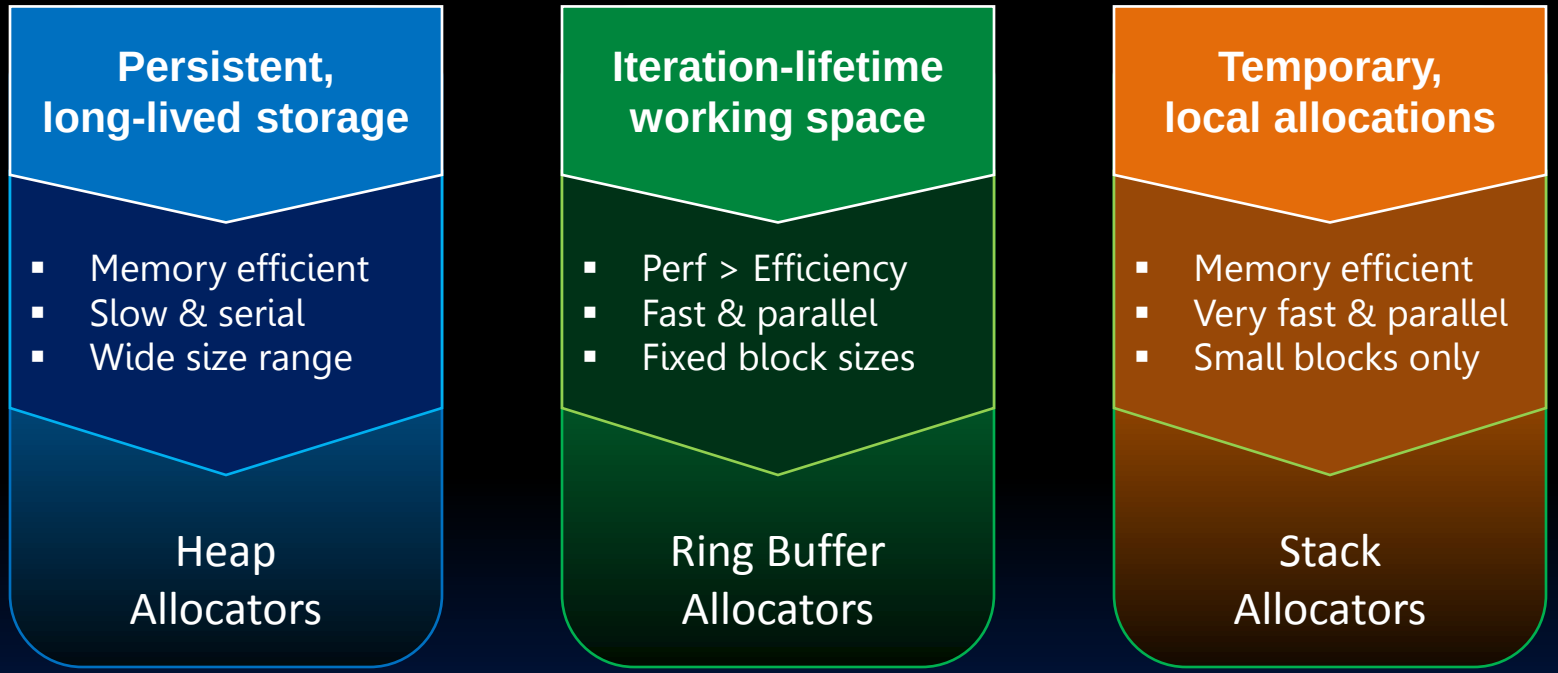


- New allocations always grows top of stack
- Free always shrinks top of stack – no fragmentation
- Very fast & simple, but requires free in reverse allocation order

Sub-Allocator Goals

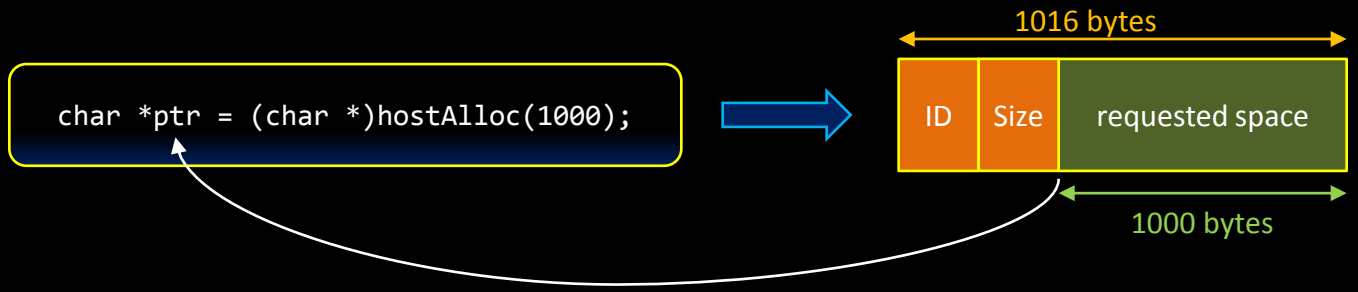
1. Fast, consistent allocation time
2. Non-blocking (i.e. no implicit synchronization)
3. Efficient – low fragmentation
4. Simple & parallelizable

Using The Right Tool For The Job



Implementing Custom Allocators

When allocating memory, record the size so you can free it



Also: Pay attention to alignment of returned pointer

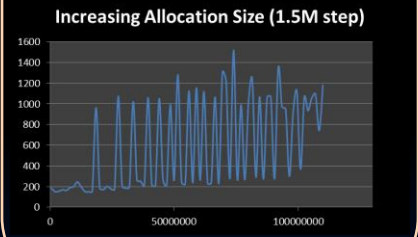
8 bytes on CPU, 256 bytes on GPU

Conclusion

Wrap new, malloc & cudaMalloc

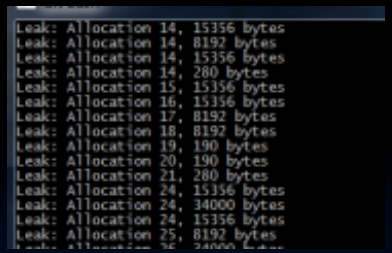
(even if internally you just call malloc/free directly)

High Performance, Non-Blocking Sub-Allocation



Host/Device Data Management

Leak Detection, Debugging & Profiling



Use The Right Type Of Allocator

(but just use malloc() for persistent heap storage)

Persistent, Long-Lived Storage

For allocations spanning multiple program iterations

- main data storage
- C++ objects & configuration data

Working Space, Lifetime Of Single Iteration

For data which does not persist outside of one iteration

- per-iteration derived quantities
- operation working space, double buffers, etc.

Temporary, Local Allocation

For transient allocations with single-procedure lifetime

- local queues, stacks & objects
- function-scope working space