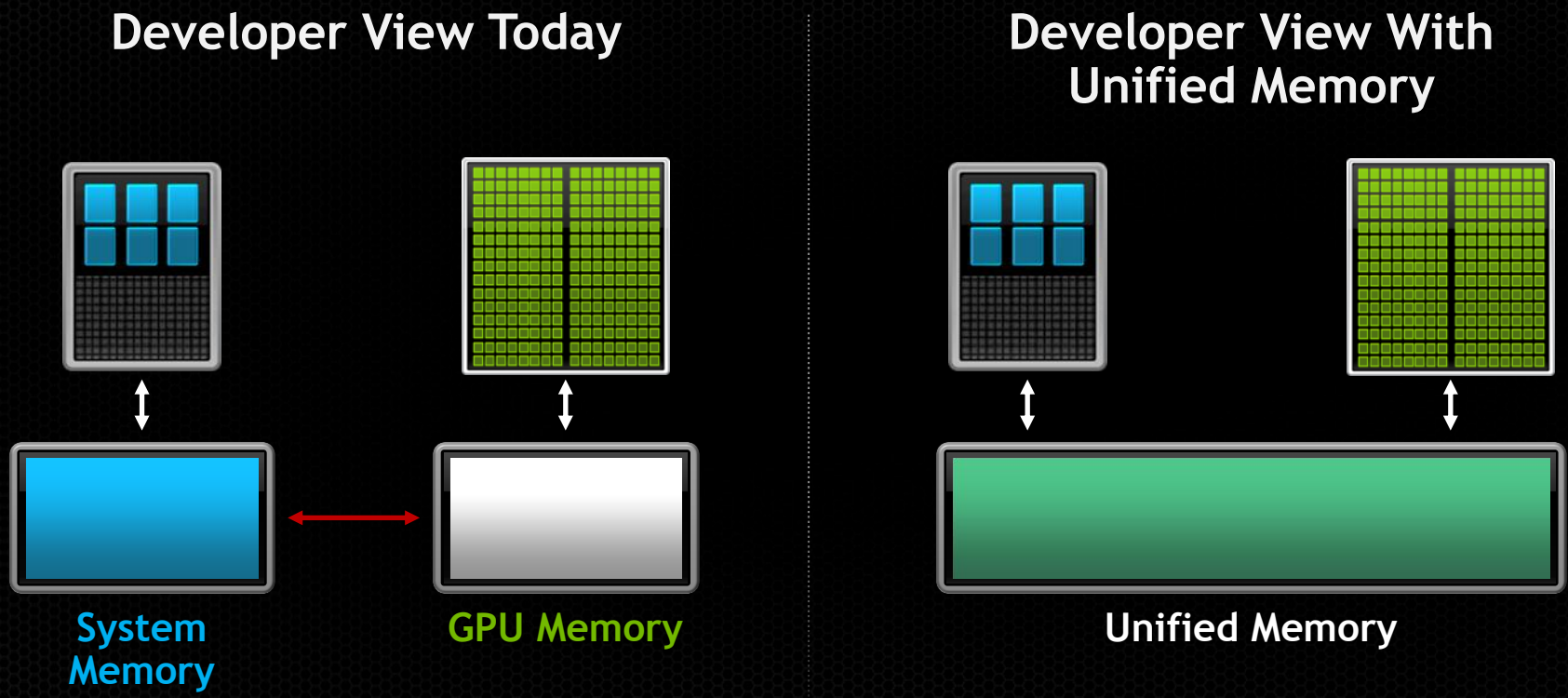
NVIDIA

UNIFIED MEMORY IN CUDA 6

MARK HARRIS

Unified Memory

Dramatically Lower Developer Effort



Super Simplified Memory Management Code

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

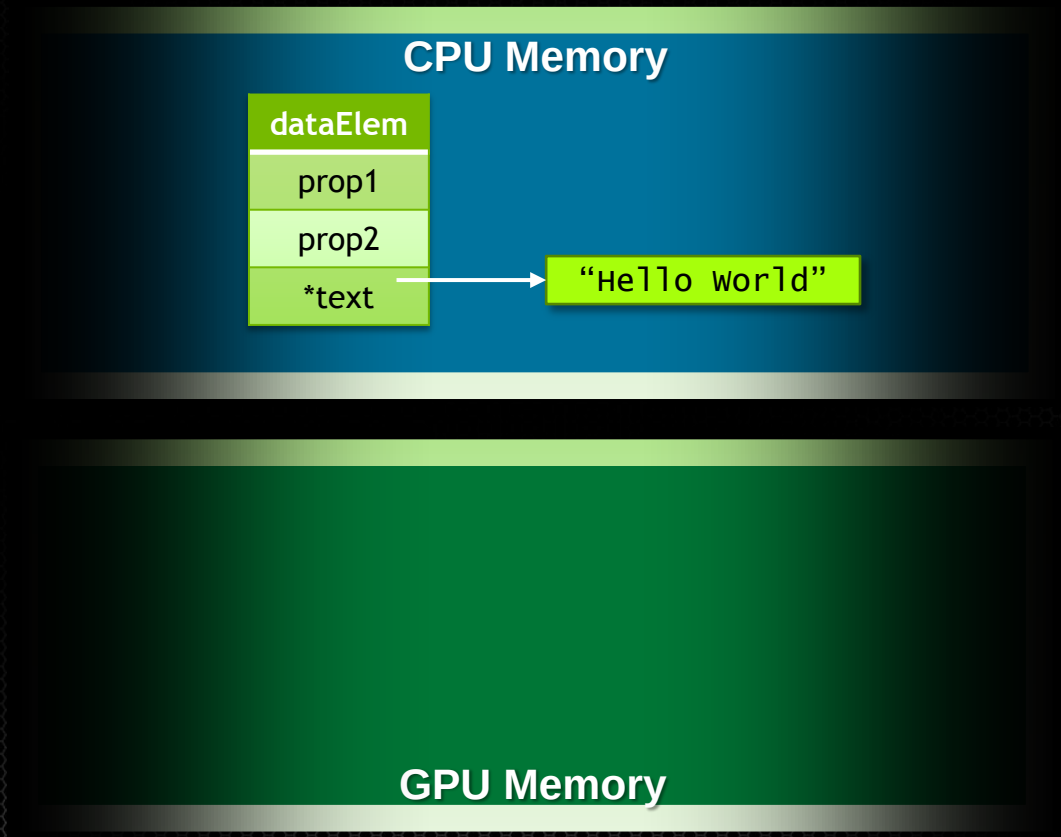
Unified Memory Delivers

1. Simpler Programming & Memory Model
   - Single pointer to data, accessible anywhere
   - Tight language integration
   - Greatly simplifies code porting

2. Performance Through Data Locality
   - Migrate data to accessing processor
   - Guarantee global coherency
   - Still allows *cudaMemcpyAsync()* hand tuning

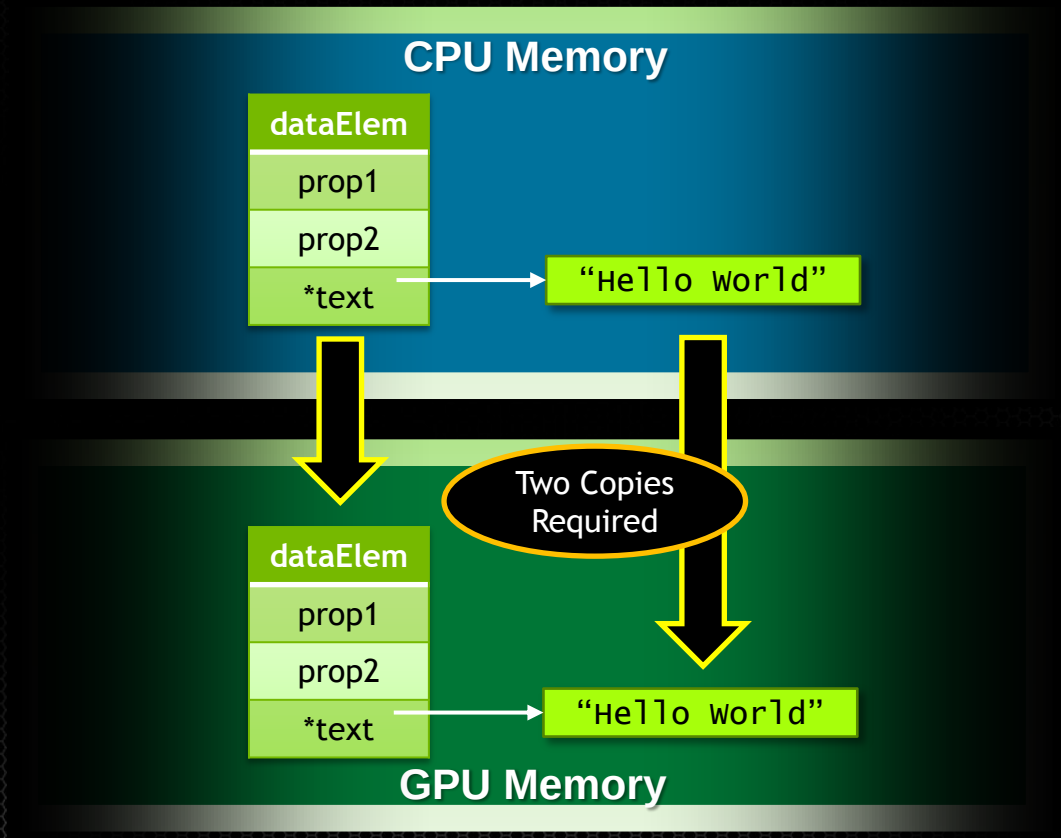
Simpler Memory Model: Eliminate Deep Copies

```
struct dataElem
{
    int prop1;
    int prop2;
    char *text;
};
```



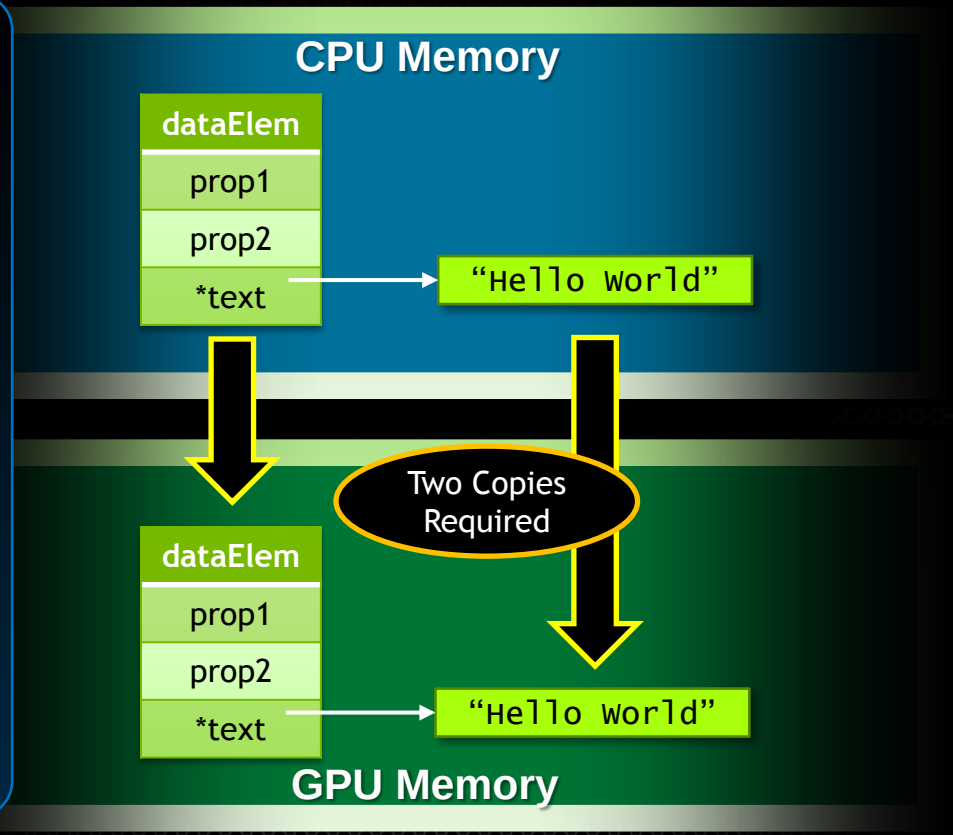
Simpler Memory Model: Eliminate Deep Copies

```
struct dataElem
{
    int prop1;
    int prop2;
    char *text;
};
```



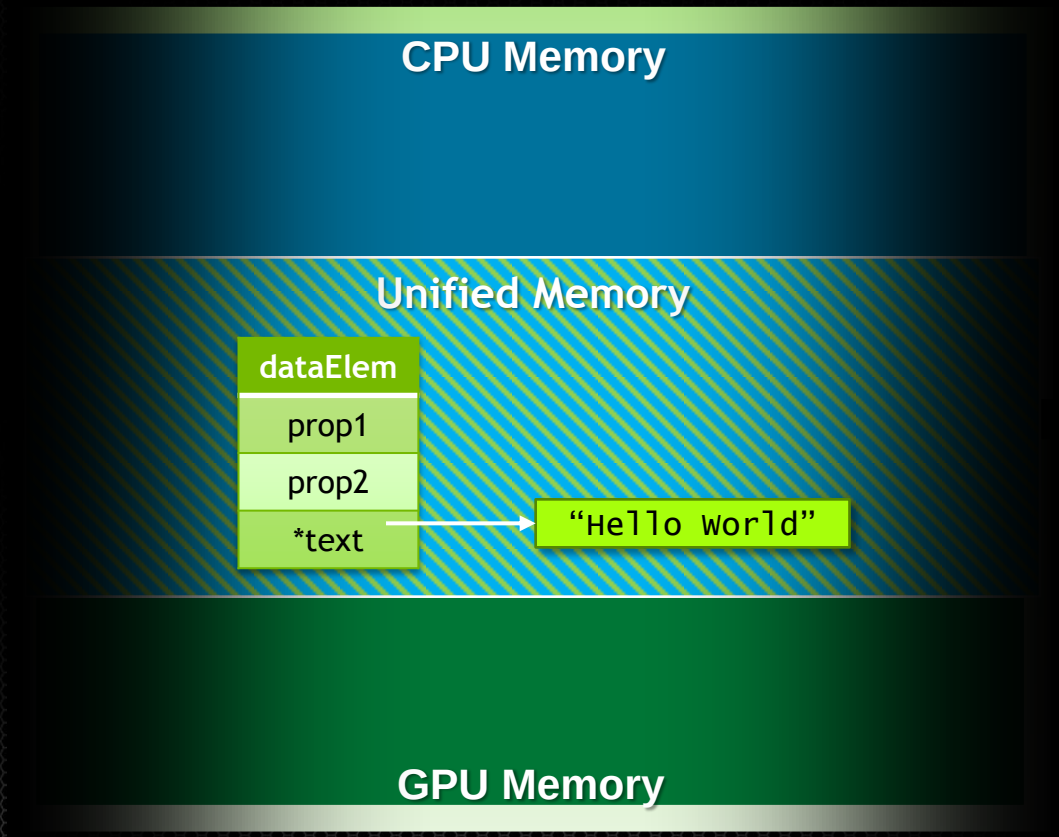
Simpler Memory Model: Eliminate Deep Copies

```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new "text" pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of "elem"
    kernel<<< ... >>>(g_elem);
}
```



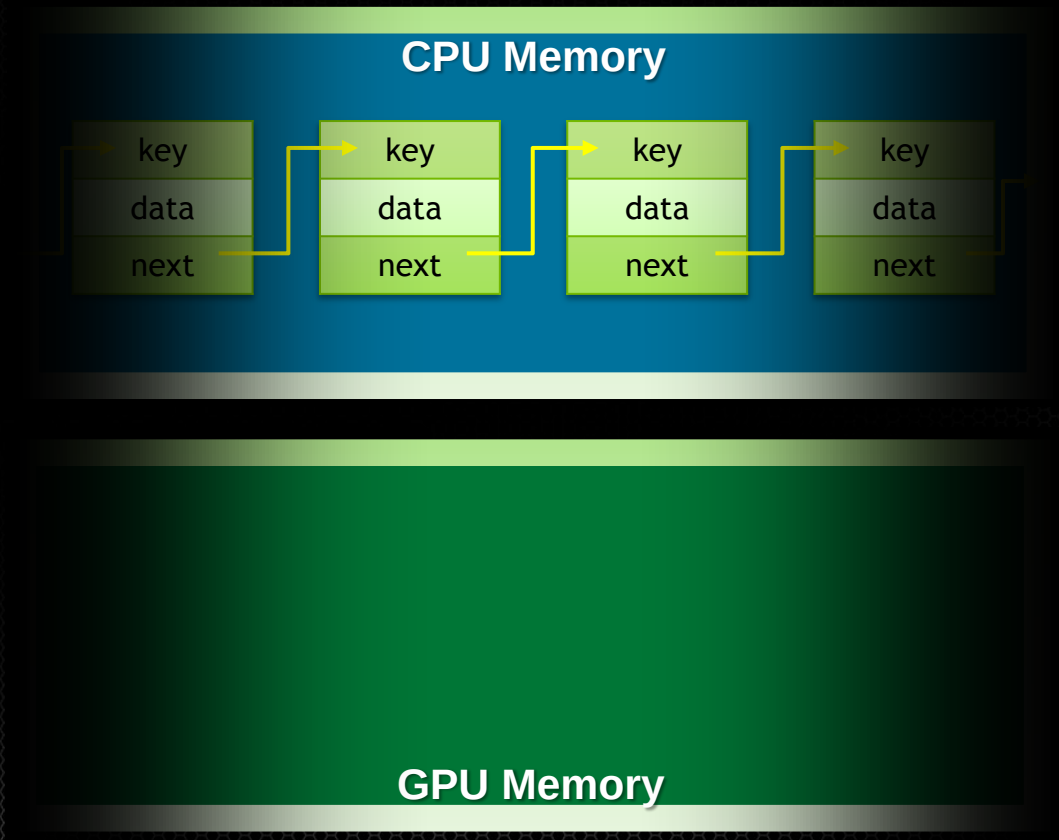
Simpler Memory Model: Eliminate Deep Copies

```
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```



Simpler Memory Model

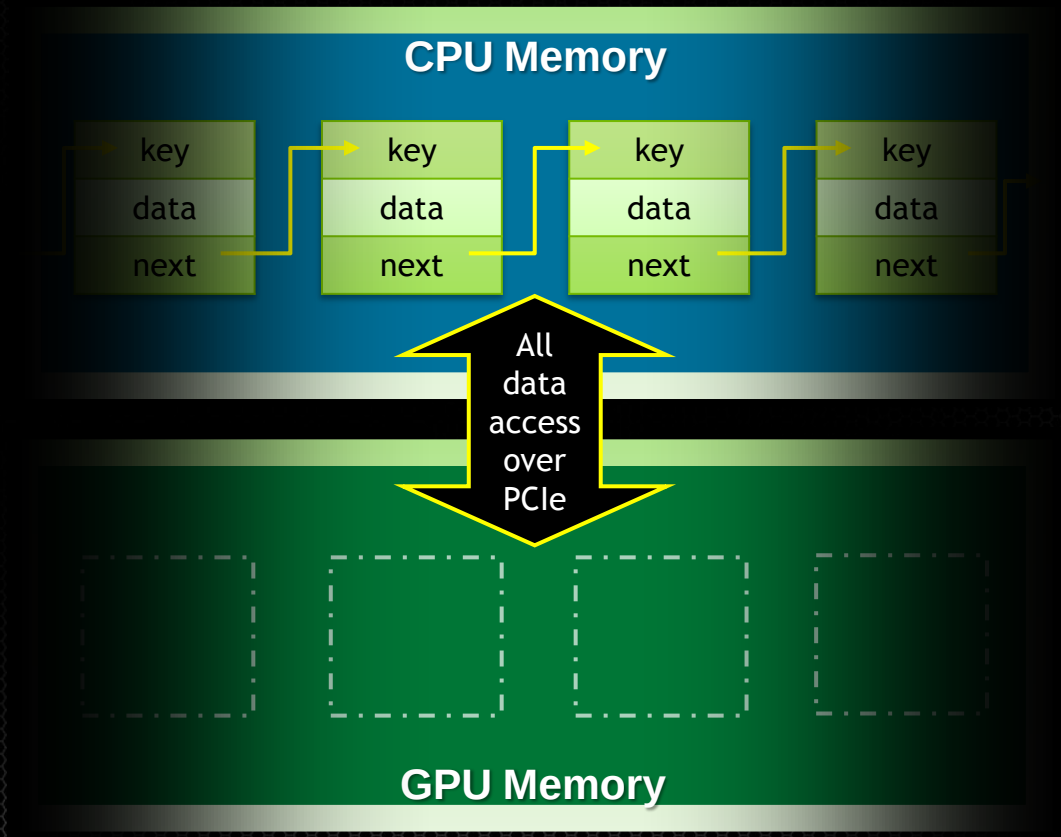
Example: GPU & CPU Shared Linked Lists



Simpler Memory Model

Example: GPU & CPU Shared Linked Lists

- Only practical option is to use zero-copy (pinned system) memory
- GPU accesses at PCIe bandwidth
- GPU accesses at very high latency

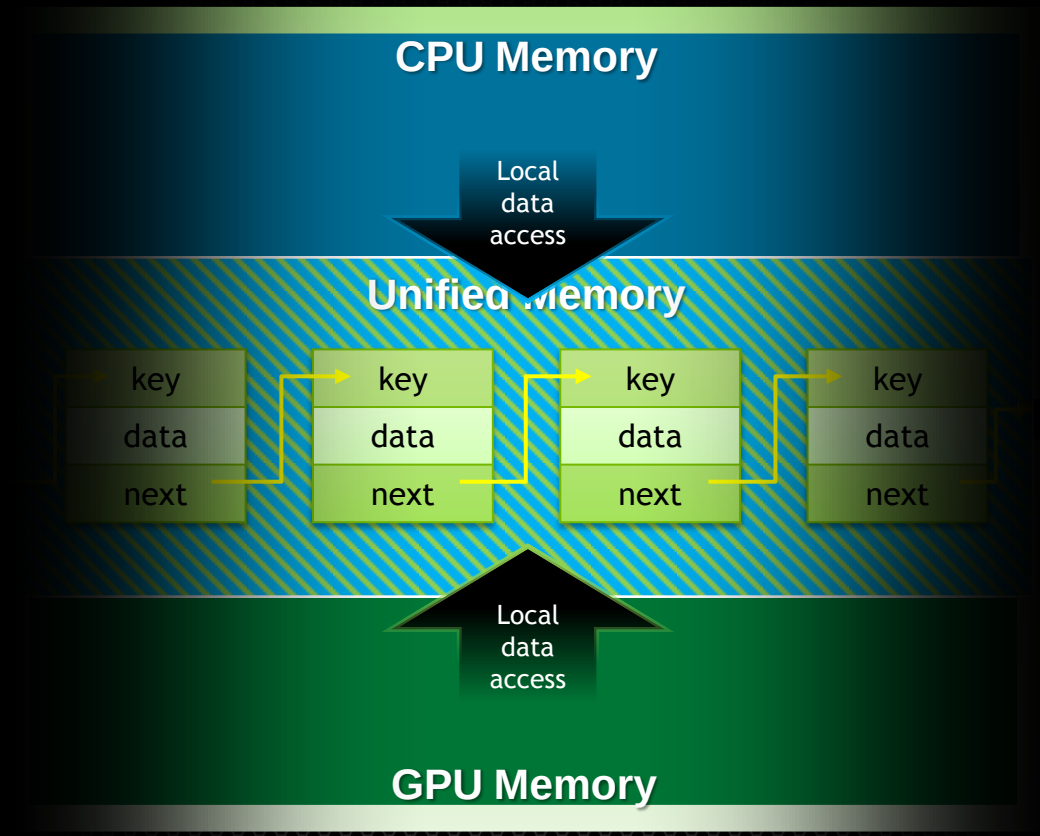


Simpler Memory Model

Example: GPU & CPU Shared Linked Lists

- Can pass list elements between Host & Device
- Can insert and delete elements from Host or Device*
- Single list - no complex synchronization

*Program must still ensure no race conditions.
Data is coherent between CPU & GPU at kernel launch & sync only

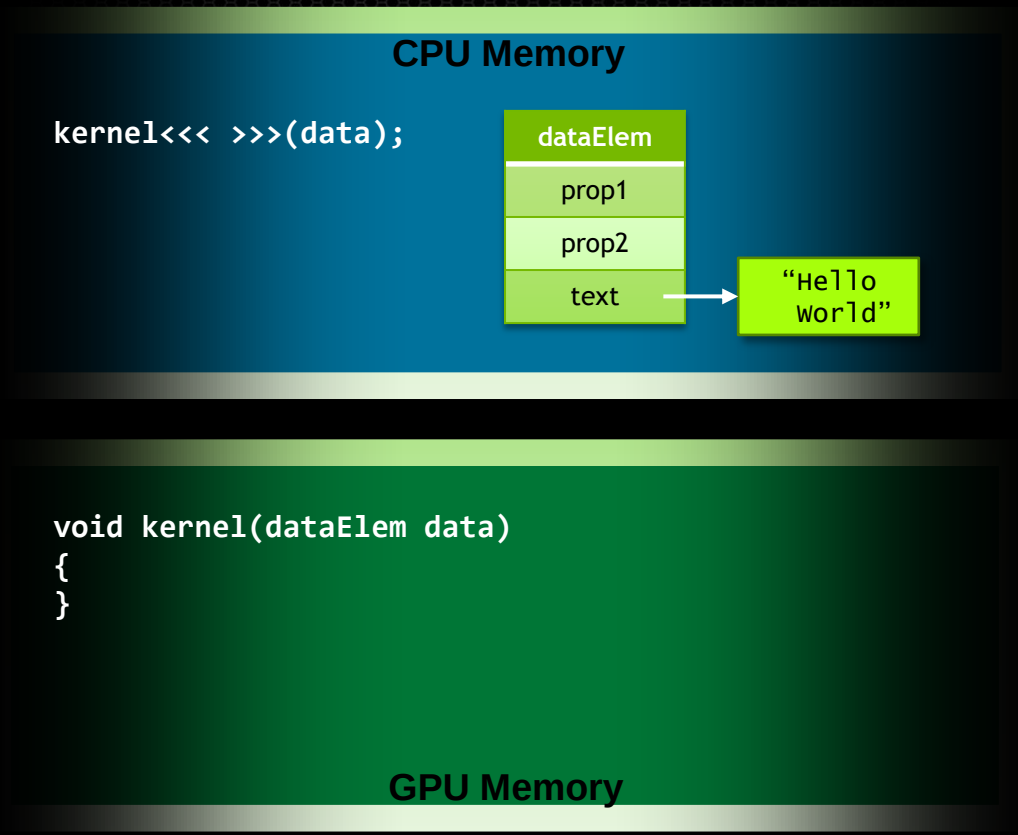


Unified Memory with C++

Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of class
class dataElem {
            int prop1;
            int prop2;
            String text;
};
```

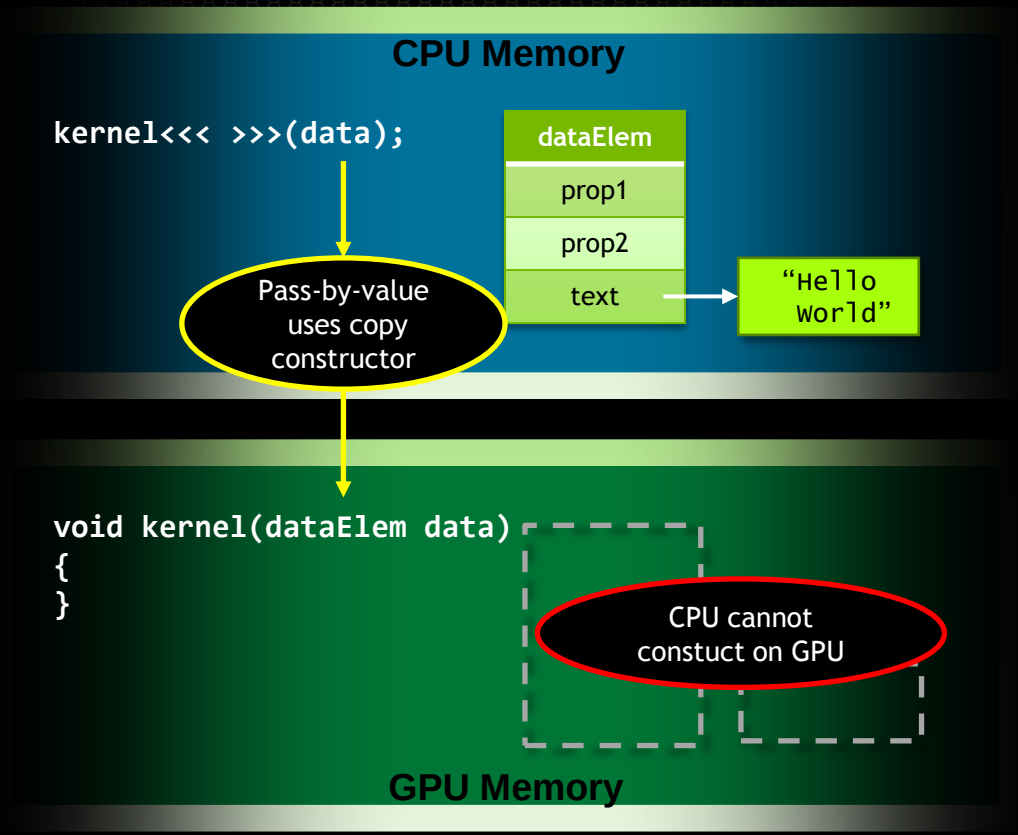


Unified Memory with C++

Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of class
class dataElem {
          int prop1;
          int prop2;
          String text;
};
```



Unified Memory with C++

C++ objects migrate easily when allocated on managed heap

- Overload *new* operator* to use C++ in unified memory region

```
class Managed {
    void *operator new(size_t len) {
        void *ptr;
        cudaMallocManaged(&ptr, len);
        return ptr;
    }

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

* (or use placement-new)

Unified Memory with C++

Pass-by-reference enabled with *new* overload

```
// Deriving from “Managed” allows pass-by-reference
class String : public Managed {
    int length;
    char *data;
};
```

NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)

Unified Memory with C++

Pass-by-value enabled by managed memory copy constructors

```
// Deriving from “Managed” allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Unified memory copy constructor allows pass-by-
value
    String (const String &s) {
        length = s.length;
        cudaMallocManaged(&data, length);
        memcpy(data, s.data, length);
    }
};
```

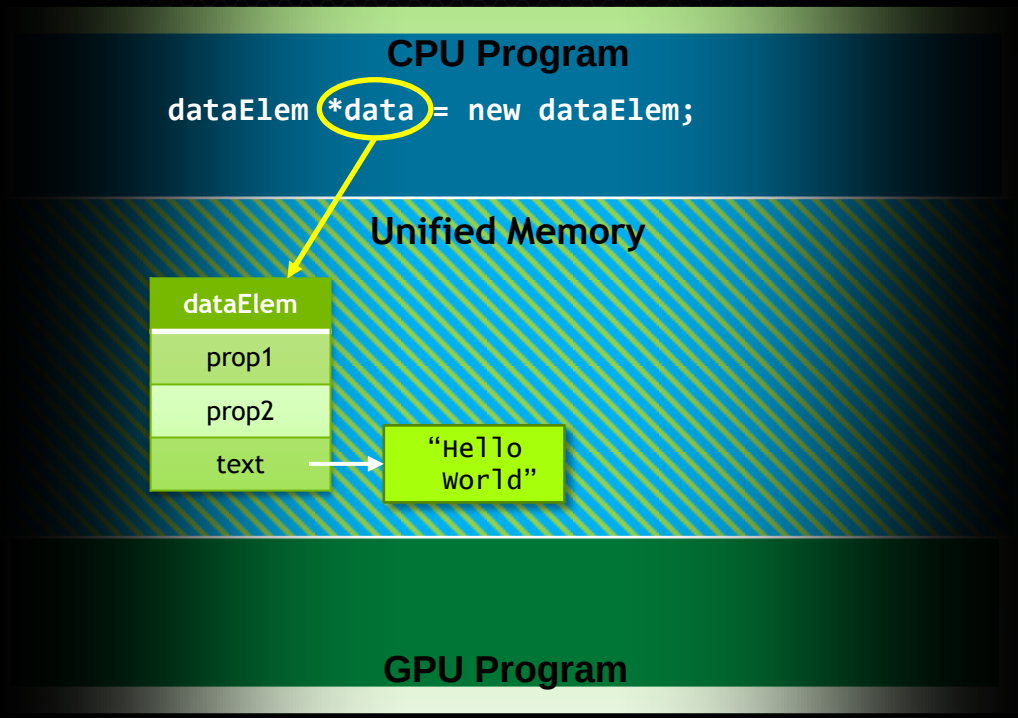
NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)

Unified Memory with C++

Combination of C++ and Unified Memory is very powerful

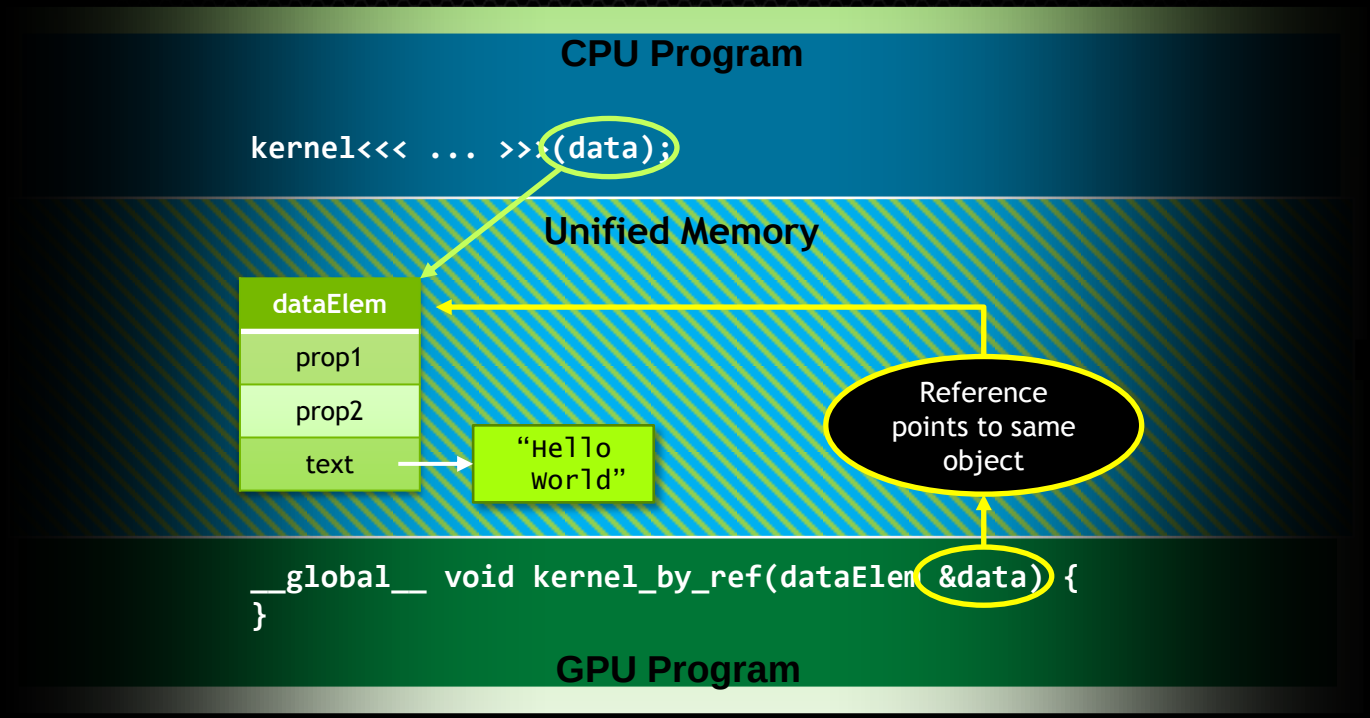
- Concise and explicit: let C++ handle deep copies
- Pass by-value or by-reference without memcpy shenanigans

```
// Note "managed" on this class, too.
// C++ now handles our deep copies
class dataElem : public Managed {
            int prop1;
            int prop2;
            String text;
};
```



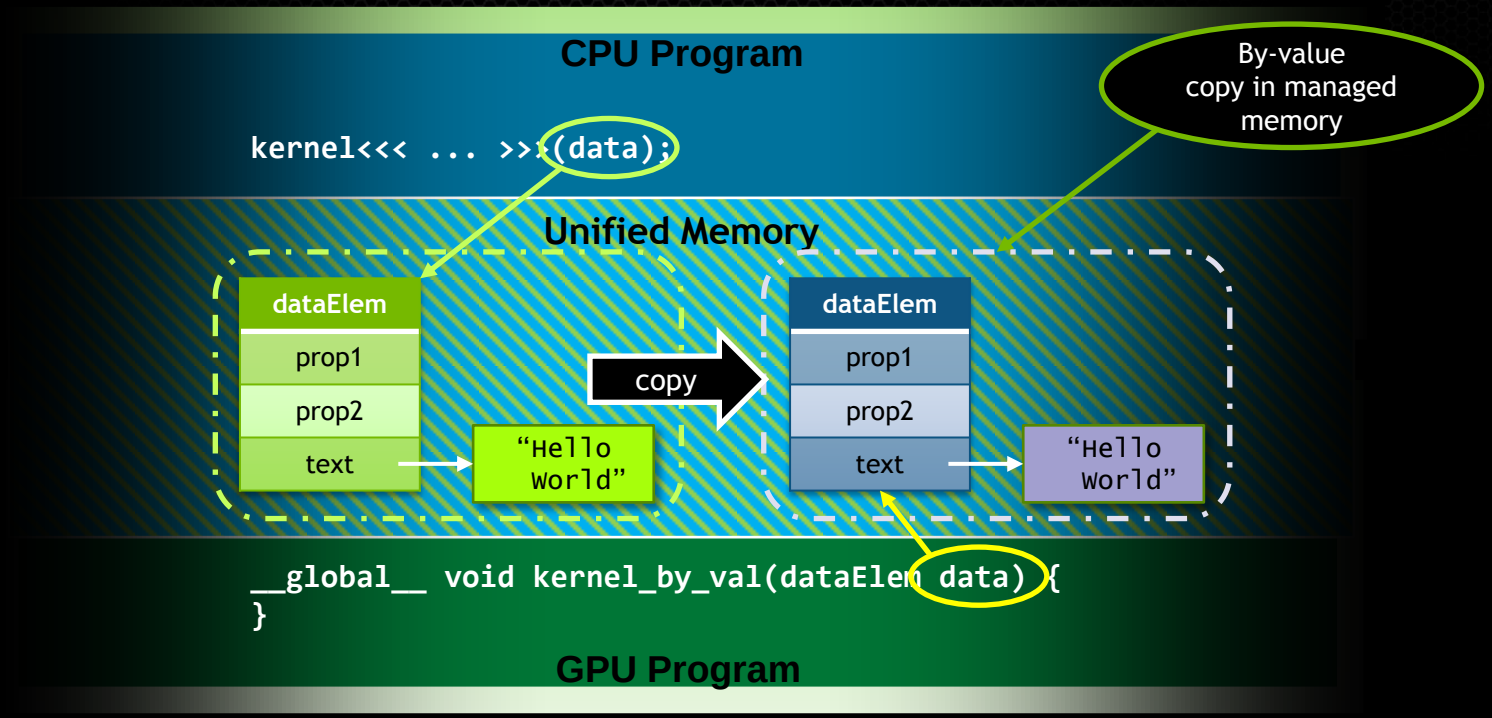
C++ Pass By Reference

Single pointer to data makes object references just work

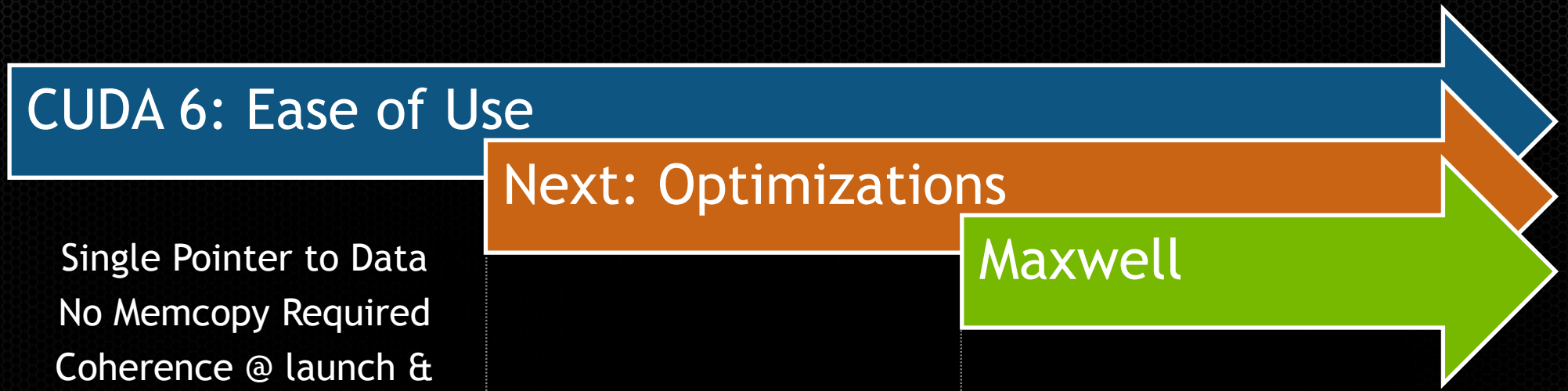


C++ Pass By Value

Copy constructors from CPU create GPU-usable objects



Unified Memory Roadmap



Single Pointer to Data

No Memcopy Required

Coherence @ launch & sync

Shared C/C++ Data Structures

Prefetching

Migration Hints

Additional OS Support

System Allocator Unified

Stack Memory Unified

HW-Accelerated Coherence

CUDA 6

1. Unified Memory
2. XT and Drop-in Libraries
3. GPUDirect RDMA in MPI
4. Developer Tools

CUDA 6

Dramatically Simplifies Parallel Programming with Unified Memory

More on Parallel Forall Blog

http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

Sign up for CUDA Registered Developer Program

https://developer.nvidia.com/cuda-toolkit