NVIDIA

CUDA UNIFIED MEMORY

Bob Crovella, 6/18/2020

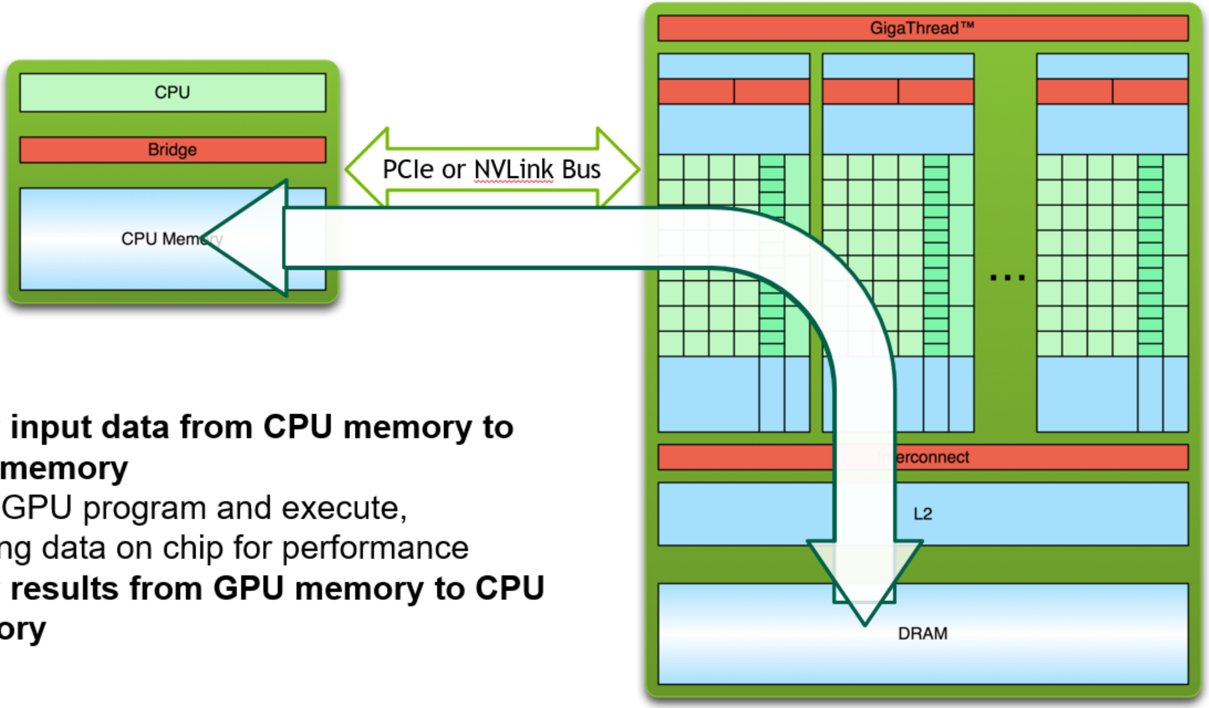
AGENDA

- Managed Memory - basic idea, objectives, benefits
- Demand-Paging, Oversubscription, Concurrency, Atomics
- Use Cases: Deep Copies, Linked Lists, C++ Objects, Graph Traversal
- Performance: Prefetching, Hints
- Multi-GPU Considerations
- Further Study
- Homework

THE CUDA 3-STEP PROCESSING SEQUENCE

Recall from Module 1…

SIMPLE PROCESSING FLOW



1. **Copy input data from CPU memory to GPU memory**
2. Load GPU program and execute, caching data on chip for performance
3. **Copy results from GPU memory to CPU memory**

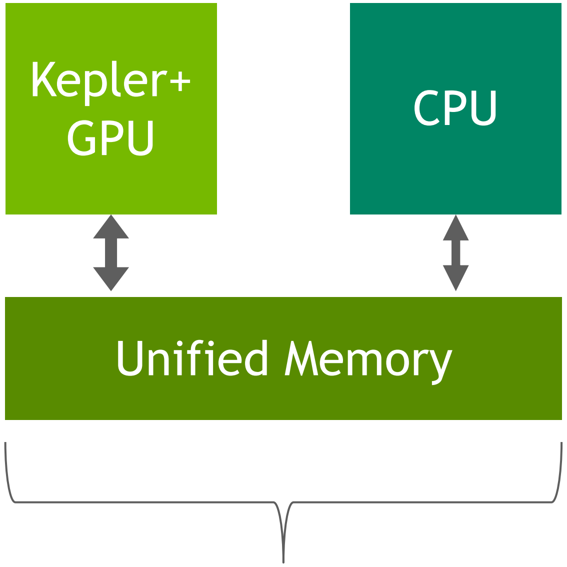
->Wouldn't it be nice if we didn't have to do (i.e. write the code for) steps 1 and 3?

INTRODUCING UNIFIED MEMORY WITH DEMAND PAGING

UNIFIED MEMORY

Reduce Developer Effort

CUDA 6+



Allocate Up To
GPU Memory Size

Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit* copy
- Simplifies code porting

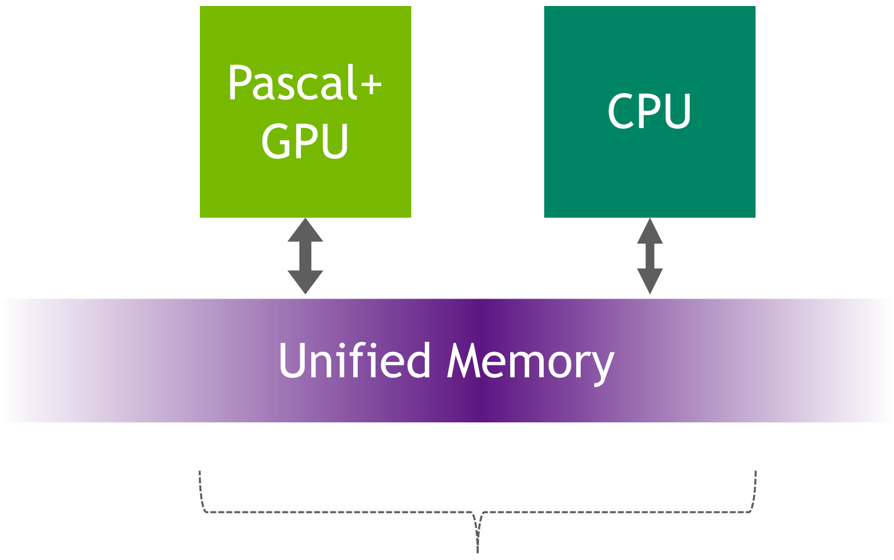
Maintain Performance through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

CUDA 8+: UNIFIED MEMORY

Demand Paging For Pascal and Beyond

CUDA 8+



Allocate Beyond GPU Memory Size

Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

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);
}
```

Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

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 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 EXAMPLE

With On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

HOW UNIFIED MEMORY WORKS ON PASCAL+

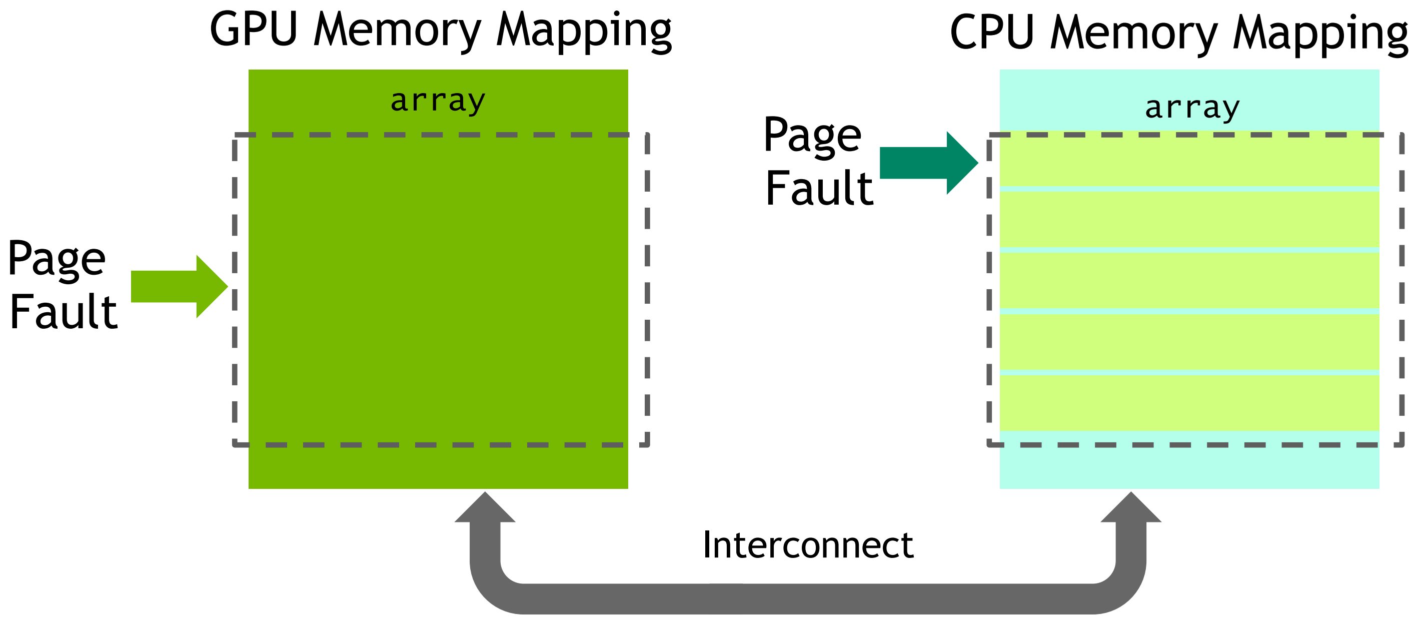
Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



ASIDE: PRE-PASCAL UM REGIME

Summary

- In effect if your device is prior to Pascal (Jetson is a special case)
- In effect if you are on windows OS (CUDA 9.x +).
- Managed data is moved en-masse at point of kernel launch (even data that your kernel may not appear to explicitly touch)
- After a kernel launch, **cudaDeviceSynchronize()** triggers the runtime to make data available to CPU code again
- No concurrent access, no on-demand migration to GPU, no oversubscription
- Just use **cudaMallocManaged()** where you would use **malloc()**, or **new**
- Use **cudaFree()** instead of **free()**, or **delete**

UNIFIED MEMORY ON PASCAL+

GPU Memory Oversubscription

```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 64 GB
  char *data;
  // be careful with size type:
  size_t size = 64ULL*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

64 GB allocation

Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU on demand.

Fails on Kepler/Maxwell

UNIFIED MEMORY ON PASCAL+

Concurrent CPU/GPU Access to Managed Memory

```
__global__ void mykernel(char *data) {
  data[1] = 'g';
}

void foo() {
  char *data;
  cudaMallocManaged(&data, 2);

  mykernel<<<...>>>(data);
  // no synchronize here
  data[0] = 'c';

  cudaFree(data);
}
```

OK on Pascal+: just a page fault

Concurrent CPU access to 'data' on previous GPUs caused a fatal segmentation fault

Note that there may still be ordering issues or data visibility issues; UM concurrency does not provide any ordering or visibility guarantees, but see system-wide atomics

UNIFIED MEMORY ON PASCAL+

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  // cpu atomic:
  __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

```
struct dataElem {
  int key;
  int len;
  char *name;
}
```

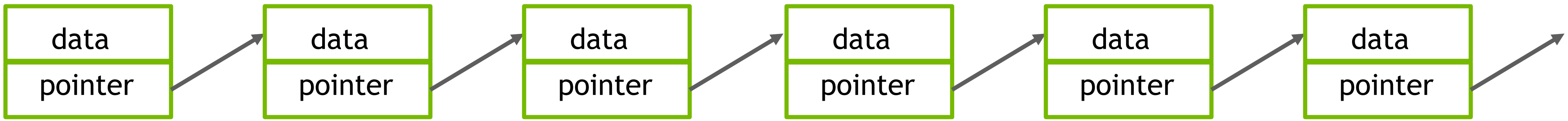
```
char buffer[len];
```

USE CASE: DEEP COPY

- Both entities (object and buffer) need to be transferred to the device
- Pointer in object needs to be “fixed” to point to new address on device for device copy of buffer

```
void launch(dataElem *elem, int N) { // an array of dataElem
  dataElem *d_elem;
  // Allocate storage for array of struct and copy array to device
  cudaMalloc(&d_elem, N*sizeof(dataElem));
  cudaMemcpy(d_elem, elem, N*sizeof(dataElem), cudaMemcpyHostToDevice);
  for (int i = 0; i < N; i++){ // allocate/fixup each buffer separately
    char *d_name;
    cudaMalloc(&d_name, elem[i].len);
    cudaMemcpy(d_name, elem[i].name, elem[i].len, cudaMemcpyHostToDevice);
    cudaMemcpy(&(d_elem[i].name), &d_name, sizeof(char *), cudaMemcpyHostToDevice);}
  // Finally we can launch our kernel
  Kernel<<< ... >>>(d_elem);}
```

USE CASE: LINKED LIST



- Similar to deep copy case
- Complex to code up the copy operation
- Unified Memory makes it trivial

USE CASE: C++ OBJECTS

Overloading **new** and **delete**

Overload new and delete in base class

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

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

inherit to build string class

```
// Deriving allows pass-by-reference
class umString : public Managed {
  int length;
  char *data;

public:
  // UM copy constructor allows
pass-by-value
  umString (const umString &s) {
    length = s.length;
    cudaMallocManaged(&data,
length);
    memcpy(data, s.data, length);
  }
};
```

USE CASE: C++ OBJECTS

Overloading **new** and **delete**

Inherit to build my class; embedded string

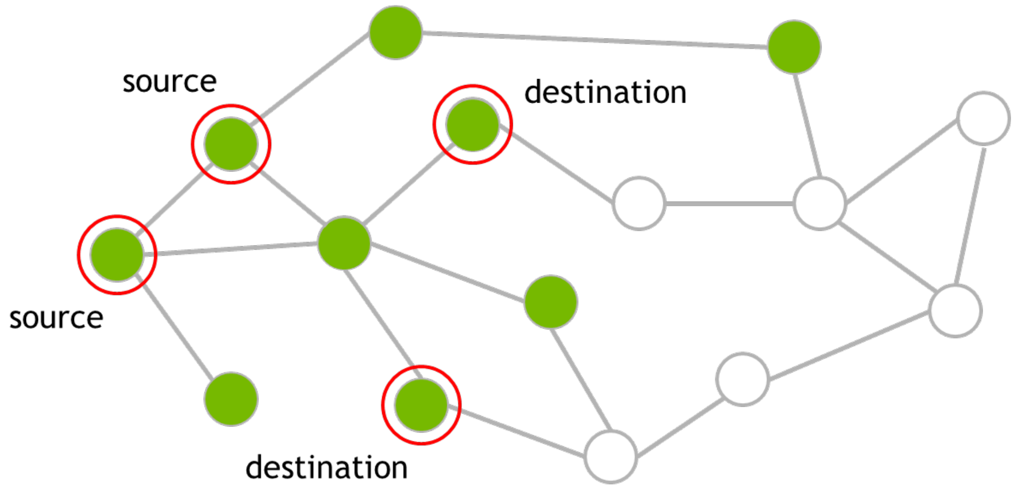
```
// Note "managed" here also

class dataElem : public Managed {
public:
  int key;
  umString name;
};
```

Profit!

```
dataElem *data = new dataElem[N];
…
// C++ now handles our deep copies
Kernel<<< ... >>>(data);}
```

USE CASE: ON-DEMAND PAGING

Graph Algorithms



PERFORMANCE TUNING ON PASCAL+

Demand Paging Impact

This kernel call runs much slower than the Pre-pascal UM 6 case, or the non-UM case.

Each page fault triggers service overhead.

Relying on page faults to move large amounts of data, page-by-page, with overhead on each page, is inefficient.

For bulk movement, a single “memcpy-like” operation is much more efficient

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
float *data;
cudaMallocManaged(&data, n*sizeof(float);
Kernel<<<256,256>>>(data);
```

PERFORMANCE TUNING ON PASCAL+

Prefetching

Explicit prefetching:

```
cudaMemPrefetchAsync(ptr, length, destDevice, stream)
```

UM alternative to cudaMemcpy(Async)

Can target any GPU and also the CPU

“Restores” performance

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
int ds = n*sizeof(float);
float *data;
cudaMallocManaged(&data, ds);
cudaMemPrefetchAsync(data, ds, 0);
Kernel<<<256,256>>>(data);
cudaMemPrefetchAsync(data, ds,
cudaCpuDeviceId); // copy back to host
```

PERFORMANCE TUNING ON PASCAL+

Explicit Memory Hints

Advise runtime on expected memory access behaviors with:

```
cudaMemAdvise(ptr, count, hint, device);
```

Hints:

`cudaMemAdviseSetReadMostly`: Specify read duplication
`cudaMemAdviseSetPreferredLocation`: suggest best location
`cudaMemAdviseSetAccessedBy`: suggest mapping

Hints don’t trigger data movement by themselves

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetReadMostly`

Data will usually be read-only

UM system will make a “local” copy of the data for each processor that touches it

If a processor writes to it, this invalidates all copies except the one written.

Device argument is ignored

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetPreferredLocation`

Suggests which processor is the best location for data

Does not automatically cause migration

Data will be migrated to the preferred processor on-demand (or if prefetched)

If possible, data (P2P) mappings will be provided when other processors touch it

If mapping is not possible, data is migrated

Volta+ adds *access counters* to help GPU make good decisions for you

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetAccessedBy`

Does not cause movement or affect location of data

Indicated processor receives a (P2P) mapping to the data

If the data is migrated, mapping is updated

Objective: provide access without incurring page faults

PERFORMANCE

Final Words

- UM is first and foremost about ease of programming and programmer productivity
- UM is not primarily a technique to make well-written CUDA codes run faster
- UM cannot do better than expertly written manual data movement, in most cases
- It can be harder to achieve expected concurrency behavior with UM.
- Misuse of UM can slow a code down dramatically
- There are scenarios where UM may enable a design pattern (e.g. graph traversal).
- Oversubscription does not easily/magically give you GPU-type performance on arbitrary datasets/algorithms
- For codes that tend to use many different libraries, each of which makes some demand on GPU memory with no regard for what other libraries are doing, UM can sometimes be a primary way to tackle this challenge (via use of oversubscription), rather than an entire rewrite of the codebase

MULTI-GPU

- Pre-Pascal Regime:
  - Allocations occur on currently selected device (just like cudaMalloc)
  - All other devices in P2P clique will receive peer mappings
  - Non-P2P: managed allocations happen in zero-copy (host) memory (performance implications!)
- Pascal+ Demand-Paging:
  - Visible to any processor on demand, with or without P2P capability/clique
  - Use prefetching/hints to guide system behavior

POWER9 NOTES

UM vs. ATS vs. HMM

- CPU/GPU dynamic memory allocations (malloc, new) can be coherently accessed among all processors
    - In particular, CPU malloc/new allocations can be read inside of a kernel instead of needing pinned memory
    - Virtual to physical address translations occur in hardware (CPU and GPU MMUs can talk to each other)
- Data is *not* migrated on-demand, but can be manually migrated with cudaMemPrefetchAsync (at lower performance than with UM)
- NVIDIA and others are working on making this functionality available more broadly through the software implementation HMM in the Linux kernel

FUTURE SESSIONS

- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

UM basics:

https://devblogs.nvidia.com/unified-memory-cuda-beginners/

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

optimization:

https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

UM architecture:

http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

Programming Guide:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

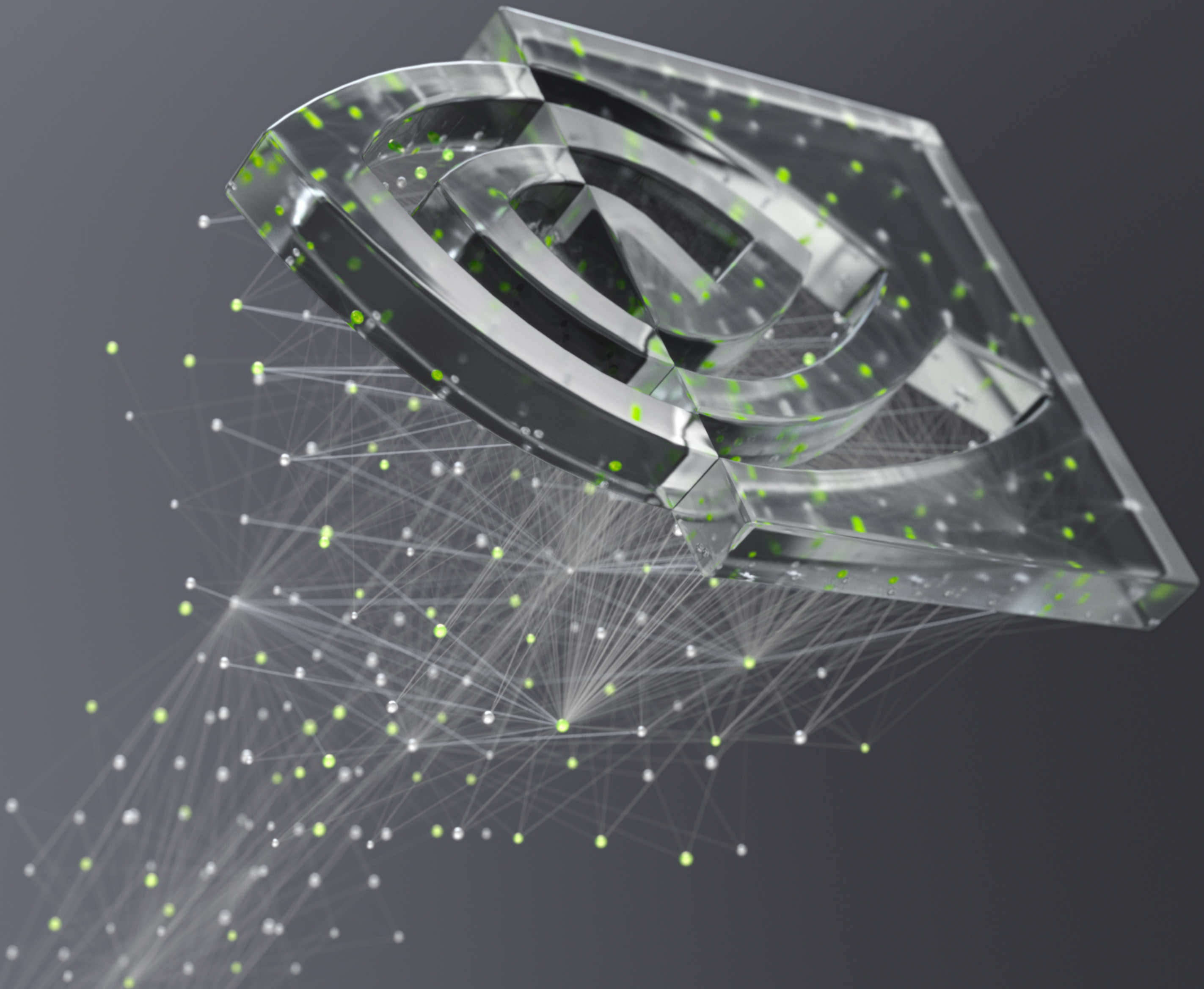
CUDA Sample Code: conjugateGradientUM

DLI: Introduction to Accelerated Computing with CUDA C++ (3 labs)

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
    - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
    - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw6/readme.md

- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA

BACKUP

HOW UNIFIED MEMORY WORKS IN CUDA 6

En-masse Movement of Data to GPU

GPU Code

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```