POLITECNICO MILANO 1863

POLITECNICO MILANO 1863 NECST laboratory

Unified Memory and (a little bit of) Multi-GPUs

Ian Di Dio Lavore - 07/05/2025

Unified Memory (UM)

Your best friend

(but also your worst enemy)

Unified Memory

Game changing paradigm introduced in CUDA 6 (current: CUDA 12.8.1)

- Before CUDA 6, the programmer handles the memories of the CPU and GPU as two different objects separated by a PCIe bus
  - Data needs to be allocated on both sides and the user will manually handle the data movements → as seen in previous lectures

- With Unified Memory the same memory region can be directly addressed by the CPUs and GPUs independently from its location.
  - Greatly simplifies the code-base
  - Allows for automatic optimizations by the CUDA execution environment

Unified-Memory

How to use it in your code:

Unified-Memory

How to use it in your code:

- `cudaMallocManaged()`
  - Behaves the same as cudaMalloc() → creates a managed memory region

Unified-Memory

How to use it in your code:

- `cudaMallocManaged()`
  - Behaves the same as cudaMalloc() → creates a managed memory region
- `cudaMemAdvise()`
  - Provides “suggestions” to the runtime on the access patterns to a specific managed memory region (we will see it later on)

Unified-Memory

How to use it in your code:

- `cudaMallocManaged()`
  - Behaves the same as cudaMalloc() → creates a managed memory region
- `cudaMemAdvise()`
  - Provides “suggestions” to the runtime on the access patterns to a specific managed memory region (we will see it later on)
- `cudaMemPrefetchAsync()`
  - Migrates data to a specific unit (GPUs or CPU) and maps it to the processor page table

Unified-Memory

How to use it in your code:

- `cudaMallocManaged()`
  - Behaves the same as cudaMalloc() → creates a managed memory region
- `cudaMemAdvise()`
  - Provides “suggestions” to the runtime on the access patterns to a specific managed memory region (we will see it later on)
- `cudaMemPrefetchAsync()`
  - Migrates data to a specific unit (GPUs or CPU) and maps it to the processor page table
- `memcpy() or cudaMemcpy()`
  - They behave in the same way (for sake of clarity in the code, use cudaMemcpy when moving managed data around)

UM → Simplifies the code

User managed memory

```
void *data;
void *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);

cpu_func1(data, N);

cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)

cpu_func3(data, N);


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

UM → Simplifies the code

User managed memory

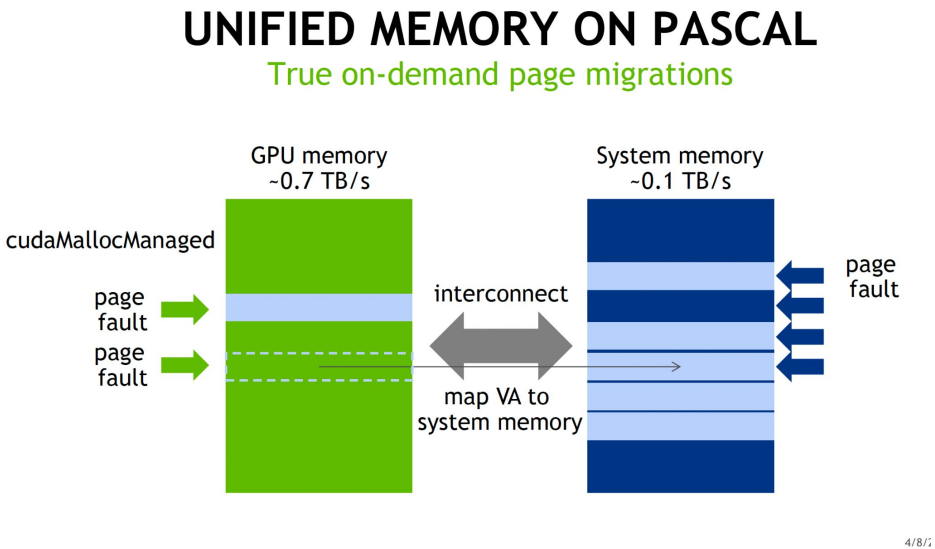
```
void *data;
void *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);

cpu_func1(data, N);

cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)

cpu_func3(data, N);


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

Unified-Memory

```
void *data;
cudaMallocManaged(&data, N);



cpu_func1(data, N);


gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);


cudaFree(data);
```

What happens under the hood?

- Unified Memory is implemented using a page-fault mechanism
- Data is moved-in/out of the devices at runtime, transparently to the user



What happens in cudaMemcpy/cudaMemset? (*full UM)

* “These systems include NVIDIA Grace Hopper, IBM Power9 + Volta, and modern Linux systems with HMM enabled …” `(concurrentManagedAccess = 1)`

`cudaMemcpy(dst, src, size, cudaMemcpyKind)`

1. When the physical location of unified memory is known, use an accurate `cudaMemcpyKind` hint.
2. Prefer `cudaMemcpyKindDefault` over an inaccurate `cudaMemcpyKind` hint.
3. Always use populated (initialized) buffers: avoid using these APIs to initialize memory.
4. Avoid using `cudaMemcpy*()` if both pointers point to System-Allocated Memory: launch a kernel or use a CPU memory copy algorithm such as `std::memcpy` instead.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#system-requirements-for-unified-memory
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#performance-tuning

Data Prefetching

cudaMemPrefetchAsync(*ptr, size, destination, stream)

- The goal is avoiding (or reducing) page-faults when a region of memory is prevalently accessed by a specific device or by the CPU
- Can be called multiple times in the life-span of your application allowing a finer granularity of data movements

```
void foo(cudaStream_t s) {
  char *data;
  cudaMallocManaged(&data, N);
  init_data(data, N);                                    // execute on CPU
  cudaMemPrefetchAsync(data, N, myGpuId, s);             // prefetch to GPU
  mykernel<<<..., s>>>(data, N, 1, compare);             // execute on GPU
  cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);     // prefetch to CPU
  cudaStreamSynchronize(s);
  use_data(data, N);
  cudaFree(data);
}
```

Without performance hints the kernel `mykernel` will fault on first access to `data` which creates additional overhead of the fault processing and generally slows down the application. By prefetching `data` in advance it is possible to avoid page faults and achieve better performance.

Data usage “hints”

Data prefetching is often insufficient when complex patterns are in place in the workload

→ cudaMemAdvise(*ptr, size, advise, device)

- Default → data migrates on first touch
- cudaMemAdviseSetReadMostly
- cudaMemAdviseSetPreferredLocation
- cudaMemAdviseSetAccessedBy

→ For each advise type you have the respective “Unset” option

Data usage “hints”

cudaMemAdviseSetReadMostly

- Lots of reads, very few writes
- device argument is ignored for this advise
- A read-only copy is created when accessing this memory region

```
char *dataPtr;
size_t dataSize = 4096;
// Allocate memory using malloc or cudaMallocManaged
dataPtr = (char *)malloc(dataSize);
// Set the advice on the memory region
cudaMemAdvise(dataPtr, dataSize, cudaMemAdviseSetReadMostly, 0);
int outerLoopIter = 0;
while (outerLoopIter < maxOuterLoopIter) {
    // The data is written to in the outer loop on the CPU
    initializeData(dataPtr, dataSize);
    // The data is made available to all GPUs by prefetching.
    // Prefetching here causes read duplication of data instead
    // of data migration
    for (int device = 0; device < maxDevices; device++) {
        cudaMemPrefetchAsync(dataPtr, dataSize, device, stream);
    }
    // The kernel only reads this data in the inner loop
    int innerLoopIter = 0;
    while (innerLoopIter < maxInnerLoopIter) {
        kernel<<<32,32>>>((const char *)dataPtr);
        innerLoopIter++;
    }
    outerLoopIter++;
 }
```

Data usage “hints”

cudaMemAdviseSetPreferredLocation

- Sets the preferred location for the data to be the memory belonging to the “device” parameter
- Either an ID of a GPU or “cudaCpuDeviceId” (an inner variable of CUDA)
- Does not move data!
- It “guides” the migration policy of the page-fault mechanism
- cudaMemPrefetchAsync will still override this “hint”

Data usage “hints”

cudaMemAdviseSetAccessedBy

- Suggests that data will be accessed by the “device” argument (GPU_id or cudaCpuDeviceId)
- It tries to keep the data always mapped in the specific processor’s page tables
- If data gets migrated (ex. cudaMemPrefetchAsync) the mappings are updated accordingly
- Useful when avoiding page-faults is relevant, more than increasing data-locality

Practical example (CUDA 12 - C Programming Guide):

> Consider for example a system containing multiple GPUs with peer-to-peer access enabled, **where the data located on one GPU is occasionally accessed by other GPUs**. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to CPU memory because the CPU cannot access GPU memory directly. Any GPU that had the `cudaMemAdviceSetAccessedBy` flag set for this data will now have its mapping updated to point to the page in CPU memory.

Extra: oversubscription

Since CUDA 8 and the Pascal or newer architectures you can oversubscribe to GPU memory with Unified Memory allocations.

You can obtain memory allocations that are larger than all available GPU memory.
The runtime system of the GPU will evict memory pages to make space for in-use active virtual memory addresses.

Ideally you could allocate all of you system (host) side memory and access it inside your GPU code.

If you are interested:

- https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/
- https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

Multi-GPU

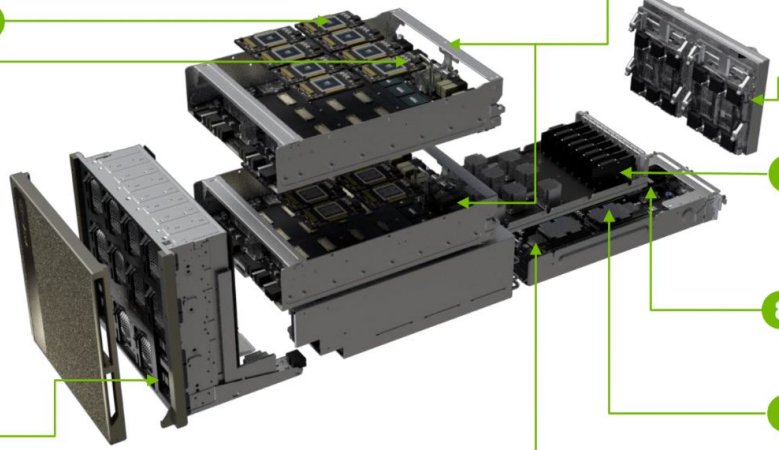
Why??

Multi-GPU systems

- LARGER
  - Work on bigger datasets (networks, fluido-dynamics, ChatGPT :D, etc)
- FASTER
  - Time-sensitive applications (weather forecast models, etc)
- COMPUTATIONAL DENSITY
  - Packing more computational power in the same space (cloud-computing)

DESIGNED TO TRAIN THE PREVIOUSLY IMPOSSIBLE

NVIDIA DGX-2



1 **NVIDIA Tesla V100 32GB**

2 **Two GPU Boards**
8 V100 32GB GPUs per board
6 NVSwitches per board
512GB Total HBM2 Memory
interconnected by
Plane Card

3 **Twelve NVSwitches**
2.4 TB/sec bi-section
bandwidth

4 **Eight EDR Infiniband/100 GigE**
1600 Gb/sec Total
Bi-directional Bandwidth

8 **Two High-Speed Ethernet**
10/25/40/100 GigE

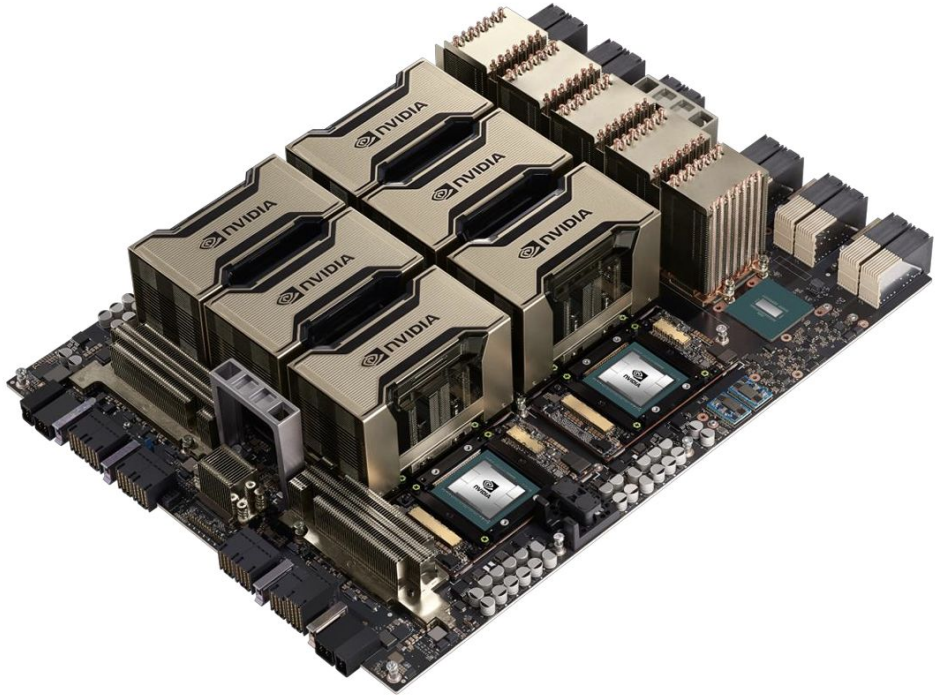
5 **Two Intel Xeon Platinum CPUs**

6 **1.5 TB System Memory**

7 **30 TB NVME SSDs**
Internal Storage

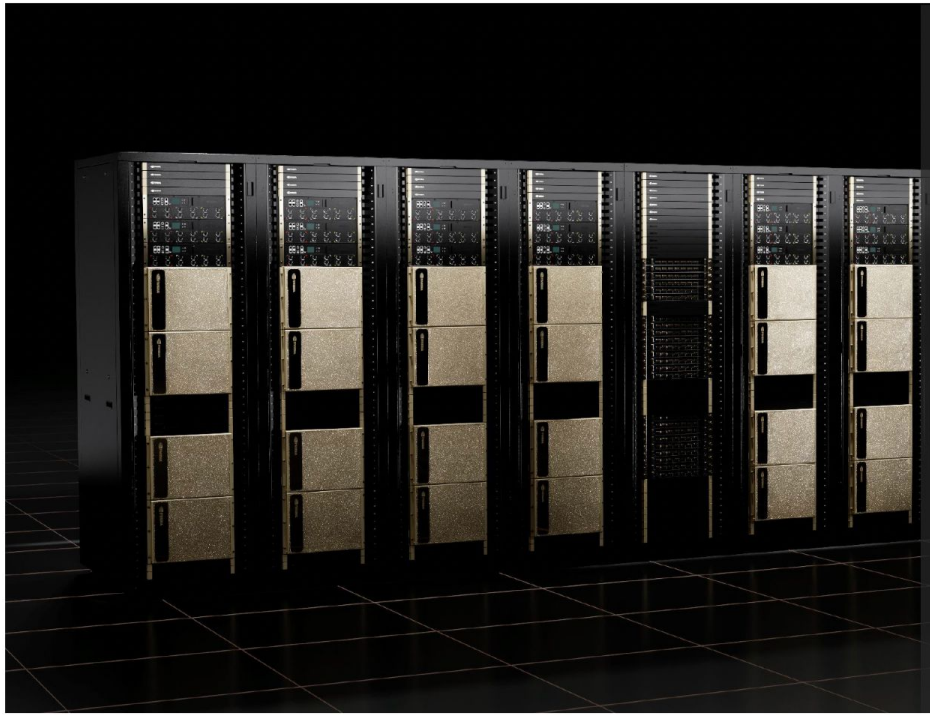
3 NVIDIA

UNPRECEDENTED ACCELERATION AT EVERY SCALE



Inside the NVIDIA Ampere Architecture - Ronny Krashinsky, Olivier Giroux, GTC March 2020

DGX H100 SUPERPOD: AI EXASCALE



DGX H100 SuperPod

- 32 DGX H100 nodes
- 256 H100 Tensor Core GPUs
- 1 ExaFLOP peak AI compute
- 70.4 TB/s bisection bandwidth
- Network optimized for AI and HPC
- New NVLink Network interconnect
- NDR 400 Gb/s InfiniBand

NVIDIA

Multi-GPU programming paradigms

- Single Threaded → CUDA
- Multi Threaded → CUDA + OpenMP* (1 GPU/Thread)
- Multi Threaded P2P → CUDA + OpenMP* (1 GPU/Thread)
- MPI → CUDA + MPI (possibility to scale on multi-node)
- NVSHMEM → CUDA + NVSHMEM (possibility to scale on multi-node)

We will cover the simplest way to handle multiple GPU → **Single Threaded** mode

*or other multithreading libraries ex. TBB

How to handle different devices?

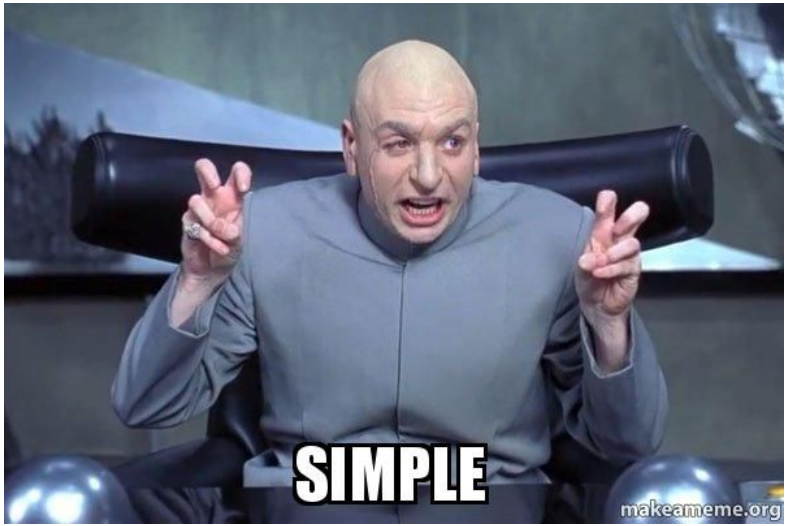
cudaSetDevice(gpu_id)

→ Switch to the context of the specified GPU

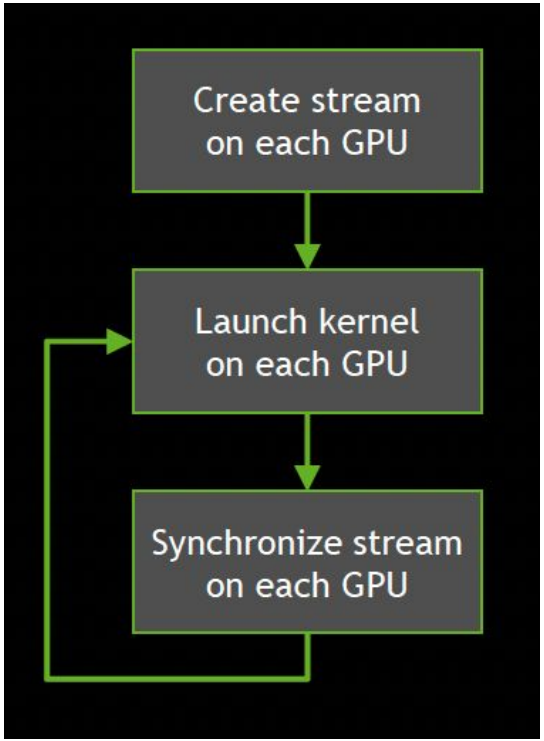
How to handle different devices?

`cudaSetDevice(gpu_id)`

→ Switch to the context of the specified GPU

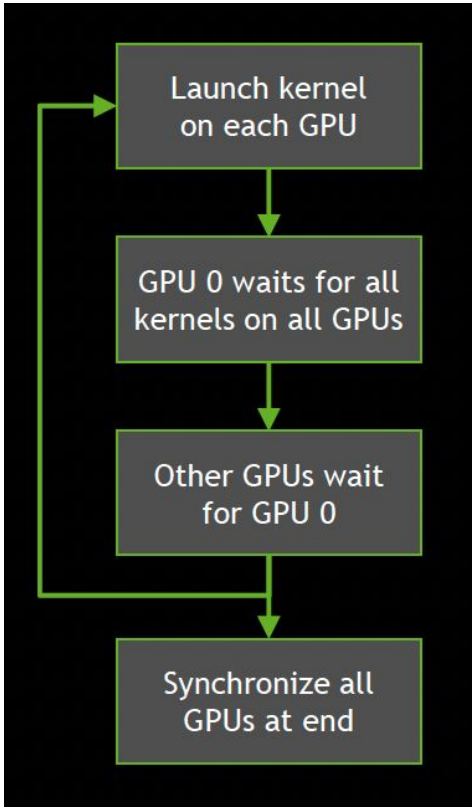


A possible way




```
// Create as many streams as I have devices
cudaStream_t stream[16];
for(int gpu=0; gpu<numGPUs; gpu++) {
       cudaSetDevice(gpu);
       cudaStreamCreate(&stream[gpu]);
}

// Launch a copy of the first kernel onto each GPU's stream
for(int gpu=0; gpu<numGPUs; gpu++) {
       cudaSetDevice(gpu);
       firstKernel<<< griddim, blockdim, 0, stream[gpu] >>>( ... );
}

// Wait for the kernel to finish on each stream
for(int gpu=0; gpu<numGPUs; gpu++)
       cudaStreamSynchronize(stream[gpu]);

// Now launch a copy of another kernel onto each GPU's stream
for(int gpu=0; gpu<numGPUs; gpu++) {
       cudaSetDevice(gpu);
       secondKernel<<< griddim2, blockdim2, 0, stream[gpu] >>>( ... );
}
```

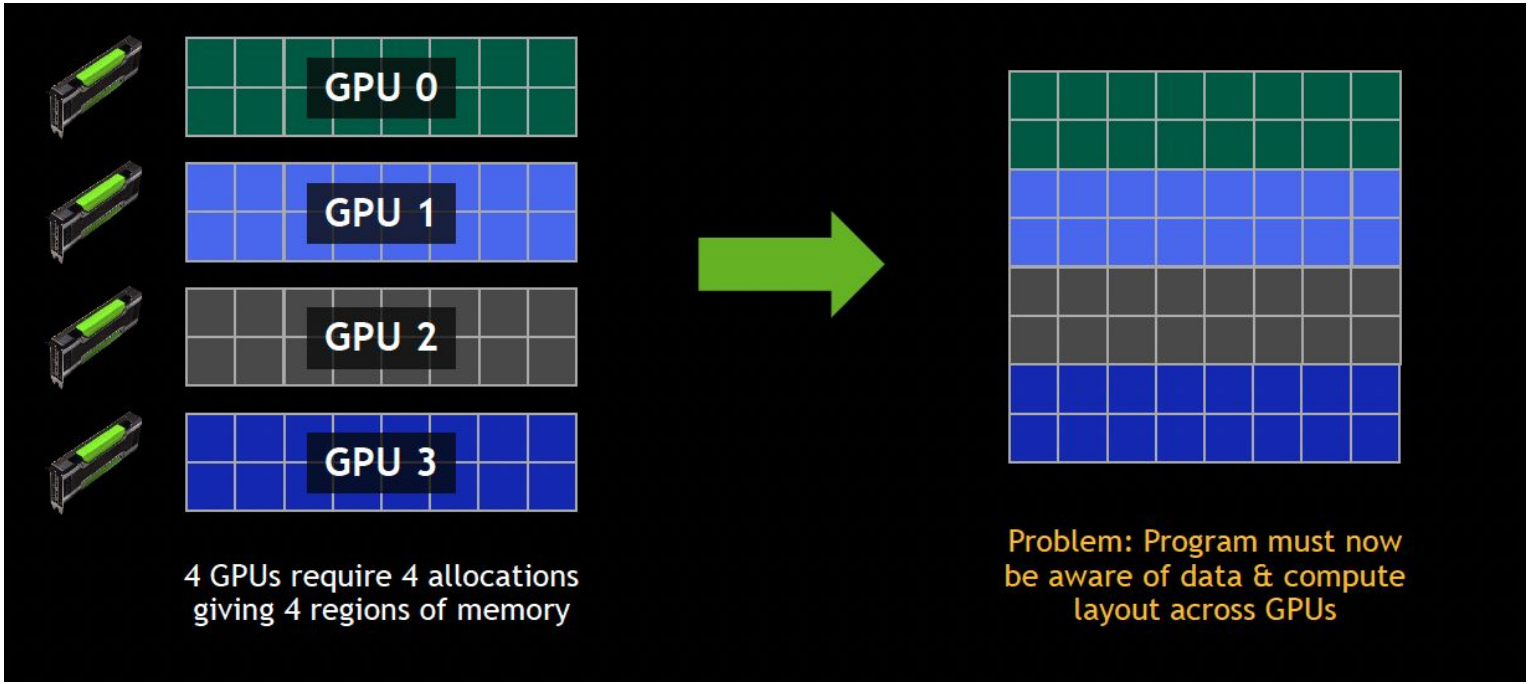
An alternative: async multi-gpu launch




```
// Launch the first kernel and an event to mark its completion
for(int gpu=0; gpu<numGPUs; gpu++) {
       cudaSetDevice(gpu);
       firstKernel<<< griddim, blockdim, 0, stream[gpu] >>>( ... );
       cudaEventRecord(event[gpu], stream[gpu]);
}

// Make GPU 0 sync with other GPUs to know when all are done
for(int gpu=1; gpu<numGPUs; gpu++)
       cudaStreamWaitEvent(stream[0], event[gpu], 0);

// Then make other GPUs sync with GPU 0 for a full handshake
cudaEventRecord(event[0], stream[0]);
for(int gpu=1; gpu<numGPUs; gpu++)
       cudaStreamWaitEvent(stream[gpu], event[0], 0);

// Now launch the next kernel with an event... and so on
for(int gpu=0; gpu<numGPUs; gpu++) {
       cudaSetDevice(gpu);
       secondKernel<<< griddim, blockdim, 0, stream[gpu] >>>( ... );
       cudaEventRecord(event[gpu], stream[gpu]);
}
```

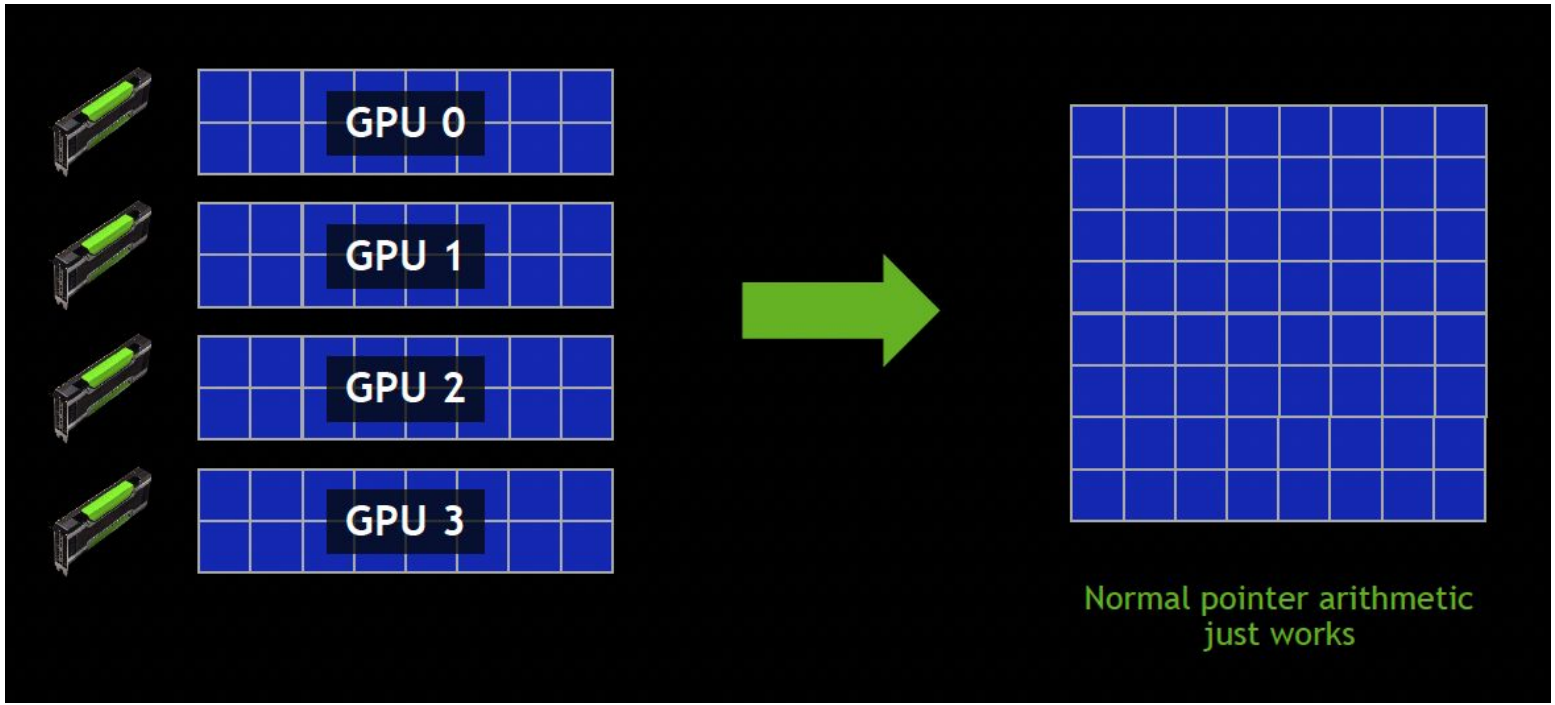
What about the memory?

cudaMalloc() creates a **partitioned** global address space

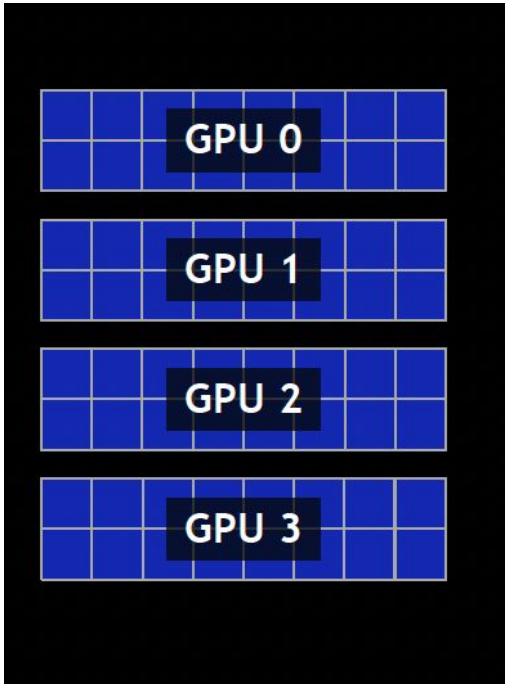


What about the memory?

cudaMallocManaged() allows one allocation to span multiple GPUs



Setting up UM in Multi-GPU systems




```
// Allocate data for cube of side “N”
float *data;
size_t size = N*N*N * sizeof(float);
cudaMallocManaged(&data, size);

// Make whole allocation visible to all GPUs
for(int gpu=0; gpu<numGPUs; gpu++) {
      cudaMemAdvise(data, size, cudaMemAdviseSetAccessedBy, gpu);
}

// Now place chunks on each GPU in a striped layout
float *start = ptr;
for(int gpu=0; gpu<numGPUs; gpu++) {
      cudaMemAdvise(start, size / numGpus,
      cudaMemAdviseSetPreferredLocation, gpu);
      cudaMemPrefetchAsync(start, size / numGpus, gpu);
      start += size / numGpus;
 }
```

Credits

Slides adapted from:

- Everything you need to know about Unified Memory, Nikolay Sakharnykh - NVIDIA
- Unified Memory on Pascal and Volta, Nikolay Sakharnykh - NVIDIA
- The future of Unified Memory, Nikolay Sakharnykh - NVIDIA
- All you need to know about programming NVIDIA's DGX-2,
  Lars Nyland & Stephen Jones, GTC 2019

Additional material:

- https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda
- https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance
- Multi GPU Programming Models - Jiri Kraus, GTC 2019
- Effective, Scalable Multi-GPU Joins,
  Tim Kaldewey, Nikolay Sakharnykh & Jiri Kraus, GTC 2019