Working with Memory

July 14, 2017

Outline

Memory Considerations

There are two primary aspects of memory management when working with accelerators:

- Since the GPU is a physically separate device connected to the host via PCI-e bus, the host and device memories have separate address spaces.
- As a throughput oriented compute architecture, the GPU device itself provides five distinctly different memory types for use in kernel development and execution.

Many features in each CUDA release are centralized around robustification of the host/device memory interaction while mastery of the various device memory types is both an art and a science essential to every GPU accelerated application.

Address Spaces in CUDA

- The physical address spaces for the CPU (host) and GPU (device) are separate.
- The CPU cannot read or write device memory on the GPU.
- The GPU cannot read or write the host memory on the CPU.
- The host application must explicitly copy data to and from the GPU device memory (using runtime API calls) in order to process it (e.g. `cudaMalloc` and `cudaMemcpy`).

Virtual Addressing: A Lightning Tour

- Physical memory locations are assigned a consecutive number.
- The standard unit of measure is the *byte* and so, for example, a memory devices of size 64 kilobytes would have memory locations 0..65535.
- The 16-bit values that specify memory locations are known as *addresses* and the process of computing addresses and operating on the corresponding memory locations is collectively known as *addressing*.
- Early computers used *physical addressing* whereby CPU instruction operands would reference physical addresses then read and write the corresponding memory locations directly.
- As software became more complex and computers running multiple jobs and/or hosting concurrent users grew more common, it became clear that allowing any program to read or write any physical memory location was suboptimal.

Virtual Addressing: A Lightning Tour

- Modern computers implement *virtual addressing* where each program (process) gets its own virtual address space.
- Using virtual addressing, processes specify virtual addresses to be translated into physical addresses by performing a series of lookups into tables setup by the operating system.
- In most systems, the virtual address space is divided into *pages* with units of addressing that are at least 4096 bytes in size.
- To perform translation, the hardware looks up a *page table entry* (PTE) that specifies the physical address where the page's memory resides.
- This address indirection allows a contiguous virtual address space to map to discontiguous pages in physical memory.

Virtual Addressing: A Lightning Tour

- When an application attempts to read or write a memory locations whose page either has not been mapped to physical memory yet or has been ejected for inactivity, the hardware signals a fault that must be handled by the operating system.
- In practice, at a minimum, the virtual address is segmented into distinct partitions: an index into a "page directory" containing many page tables and an index into the page-table specified by the first index.
- This type of hierarchical design reduces the amount of memory needed for the page tables and enables inactive page tables to be marked nonresident and swapped to disk, much like inactive pages of memory.

Virtual Addressing: A Lightning Tour

- Address translation is performed on *every* memory access performed by the CPU.
- To optimize address translation, the CPU contains special hardware caches called translation lookaside buffers (TLBs) and "page walkers" that resolve cache misses in the TLBs by reading the page tables.
- Today CPUs include hardware support for "unified address spaces" where multiple CPUs can access each others memory efficiently. This hardware mechanism is called Hyper Transport (HT) on AMD chips and QuickPath Interconnect (QPI) on Intel chips.
- Since these hardware facilities enable CPUs to access any memory location in the system, this allows reference to "the CPU" and "CPU address space" irrespective of the number of CPUs in the system.

Virtualized Host Memory

- On all operating systems that run CUDA, host memory is *virtualized*. The operating system component that manages virtual memory is called the *virtual memory manager* or VMM.
- Physical pages of memory, usually 4KB or 8KB in size, can be relocated without changing their virtual address.
- In particular, these pages can be swapped to disk – effectively enabling the system to have more virtual memory than physical memory.
- When a page is marked "nonresident", an attempt to access the page will signal a page fault to the operating system.
- The page fault signal prompts the OS to find a physical page avaliable to copy the data from disk and resume execution with the virtual page pointing to the new physical location.

Host Memory: Direct Access

- The VMM monitors memory activity and uses heuristics to decide when to "evict" pages to disk and resolves the pages faults that happen when evicted pages are referenced.
- Additionally, the VMM provides services to hardware drivers to facilitate direct access of host memory by the hardware.
- Many peripherals such as disk controllers, network controllers, and GPUs read and/or write host memory using a VMM provided facility known as *direct memory access* (DMA).
- DMA enables the hardware to operate concurrently with the CPU. Whatever the peripheral device, without DMA, the device driver must use the CPU to copy data to or from special hardware buffers.

Host Memory: Page-Locking

- To facilitate DMA, the VMM provides a service called *page-locking* whereby the VMM marks pages as ineligible for eviction and therefore the physical address can not change.
- Once memory is page-locked, drivers can program their DMA hardware to reference the physical addresses of that memory.
- Note that page-locking memory makes the underlying physical memory unavailable for other uses by the operating system.
- Page-locking a lot of memory can have adverse effects on system performance.
- Page-locked memory is sometimes known as *pinned* memory. The idea is that since the OS can not change the physical address then that memory has been "pinned in place".

Host Memory: Pinned Memory

- From the OS point of view, the terms *pinned* and *page-locked* are synonymous.
- However, from the CUDA point of view, *pinned* memory is page-locked but also tracked by the device driver and mapped for access by the hardware.
- In CUDA, pinned memory is allocated and freed using `cudaHostAlloc()` and `cudaHostFree()`. Host memory allocated this way is paged-locked and configured for DMA by the current CUDA context.
- The CUDA driver tracks the memory ranges allocated in this way and automatically accelerates memcpy operations that reference pinned memory.
- Asynchronous memcpy operations only work with pinned memory. GPUs cannot access pageable memory at all.

Host Memory: Mapped Pinned Memory

- By default, pinned memory is not mapped into the CUDA address space and is not accessible within a CUDA kernel.
- Memory that has been both pinned and *mapped* into the address space of the CUDA context allows CUDA kernels to both read and write those memory addresses.
- For integrated GPUs, mapped pinned memory enables *zero copy*: Since the host (CPU) and device (GPU) share the same memory pool, they can exchange data without explicit copies.
- On discrete GPUs, mapped pinned memory is advantageous only in certain cases with small amounts of data since the data is not cached on the GPU.
- As of CUDA 4.0, if *unified virtual addressing* (UVA) is in effect, all pinned host allocations are mapped.

Unified Virtual Addressing

- A feature called *unified virtual addressing* (UVA) was introduced in CUDA 4.0.
- When UVA is in force, the CUDA driver allocates memory for both CPUs and GPUs from the same virtual address space.
- The CUDA driver accomplishes UVA by having its initialization routine perform large virtual allocations from the CPU address space and then mapping subsequent GPU allocations into those address ranges. Note that these large initial virtual allocations are not backed by physical memory.
- Under UVA, for mapped pinned allocations, the GPU and CPU pointers are the same. For other types of allocation, the driver can infer the device for which a given allocation was performed from the address.
- However, even when UVA is in effect, the CPU(s) cannot access device memory and, by default, the GPU(s) cannot access one another's memory.

Notes on Unified Virtual Addressing

- UVA is supported on all 64-bit platforms and 64-bit Windows via the TCC driver since the WDDM driver does not yet support UVA.
- The TCC driver is available only with enterprise Tesla compute GPUs and can be enabled using the `nvidia-smi` utility.
- Since x64 CPUs support 48-bit virtual address spaces (256 terabytes), while CUDA enabled GPUs only support 40 bit, applications using UVA should make sure CUDA gets initialized early to guard against CPU code using virtual address needed by CUDA.

Peer-to-Peer Memory Mapping

- UVA is also a necessary precondition for enabling peer-to-peer (P2P) transfer of data directly across the PCIe bus effectively bypassing host memory.
- Because UVA is always in force when using peer-to-peer access, the address ranges for different devices do not overlap, and the driver (and runtime) can infer the owning device from the pointer values.
- Peer-to-peer mappings of device memory enable a kernel running on one GPU to read and/or write memory that resides in another GPU.
- Since the GPUs can only "talk" at PCI-e rates, GPUs should only exchange modest amounts of data.
- However, the advent of NVLINK allows GPUs to exchange data with much higher bandwidth enabling more diverse peer-to-peer interactions.

Peer-to-Peer Memory Mapping

- For P2P addressing to work, the following conditions apply:
  - Unified virtual addressing (UVA) must be in effect.
  - The GPUs must be SM 2.x or higher.
  - The GPUs must be based on the same chip set
  - The GPUs must be on the same I/O hub.
- Peer-to-Peer access uses a small amount of extra memory to hold more page tables and makes memory allocation more expensive, since the memory must be mapped for all participating devices.
- Note that peer-to-peer mappings are thus not enabled automatically. P2P must be specifically requested by calling `cudaDeviceEnablePeerAccess()`.
- Once peer-to-peer access has been enabled, all memory in the peer device, including new allocations, is accessible to the current device until `cudaDeviceDisablePeerAccess()` is called.

Unified Managed Memory

Here we reach the full arc of our journey through the host and device memory interactions within CUDA. Unified Memory represents the forefront of the CUDA programing model and paves the way for exciting features in the upcoming release of CUDA 8.

- *Unified Memory* was introduced in CUDA 6.0 with the intent to simplified memory management by automatically migrating the necessary data back and forth between host and device – eliminating explicit memcpy calls and deep copy issues.
- In a nutshell, the driver automatically migrates data between GPU and CPU, so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.
- Unified memory is effectively all the programatic benefits of UVA zero-copy but scalable since the driver will automatically migrate the necessary data, at the level of individual pages, between host and device memory.

Unified Managed Memory

- The term *managed memory* refers to memory allocated dynamically on the host via `cudaMallocManaged()` or statically declared in global scope using the `__managed__` annotation:

```
1  __device__ __managed__ int y[N];
```

- It is very important to keep in mind that managed memory is visible to all available devices in the system. To constrain which devices are visible/exposed to the application, set the environment variable `CUDA_VISIBLE_DEVICES`.
- Managed memory is *interoperable* with un-managed device-specific memory allocations such as `cudaMalloc()`. A kernel can utilize both managed and un-managed memory.

Unified Managed Memory: Coherence

- Unified Memory attempts to optimize memory performance by migrating data towards the device where it is being accessed.
- The physical location of data is invisible to the program and may be changed at any time, but accesses to the virtual address of the data will remain valid and coherent from any processor regardless of locality.
- Note that maintaining coherence is the primary requirement ahead of performance. Within the constraints of the host operating system, the unified memory management system can either fail accesses or move data in order to maintain global coherence between processors.
- The memory system will try to place data in a location where it can most efficiently be accessed without violating coherency.

Unified Memory: Exclusive Access

- To help ensure coherency, the Unified Memory programming model puts constraints on data accesses while both the CPU and GPU are executing concurrently.
- The device has exclusive access to all managed data while any kernel operation is executing, regardless of whether the specific kernel is actively using the data.
- Concurrent host/device accesses, even to different managed memory allocations, will cause a segmentation fault because the page is considered inaccessible to the host.
- The behavior of dynamically allocating memory, with `cudaMallocManaged()` or `cuMemAllocManaged()`, while the device is active is unspecified until additional work is launched or the device is synchronized. Attempting to access that memory on the host during this time may or may not cause a segmentation fault.

Unified Memory: Exclusive Access

- To help ensure coherency, the Unified Memory programming model puts constraints on data accesses while both the CPU and GPU are executing concurrently.
- The device has exclusive access to all managed data while any kernel operation is executing, regardless of whether the specific kernel is actively using the data.
- Concurrent host/device accesses, even to different managed memory allocations, will cause a segmentation fault because the page is considered inaccessible to the host.
- The behavior of dynamically allocating memory, with `cudaMallocManaged()` or `cuMemAllocManaged()`, while the device is active is unspecified until additional work is launched or the device is synchronized. Attempting to access that memory on the host during this time may or may not cause a segmentation fault.

Unified Managed Memory: Multi-GPU

- Managed allocations are automatically visible to all GPUs in a system via the peer-to-peer capabilities.
- While multi-GPU systems are able to use managed memory, data does not migrate (i.e. get copied) between GPUs.
- Managed memory allocation behaves similarly to unmanaged memory allocations whereby the current active device is the home for the physical allocation and all other GPUs receive peer mappings to the memory (i.e. it's just UVA zero-copy between two devices over PCI-e).
- If peer mappings are not available, perhaps due to GPUs of different architectures, then the system will fall back to using zero-copy *host* memory in order to guarantee data visibility. That is, **double** zero-copy $GPU(i) \rightarrow host \rightarrow GPU(j)$ over PCI-e (don't do this). More details in the docs.

Unified Managed Memory: Profiling

- Both **nvprof** and **nvvp** have been updated to provide detailed Unified Memory profiling capabilities.
- Use **nvprof** `--profile-api-trace runtime` to get details on runtime-api calls for managed memory
- Use **nvprof** `--print-gpu-trace` to print individual kernel invocations (including CUDA memcpy's/memset's) and sort them in chronological order. In particular, looking for Unified Memory Memcpy HtoD and DtoH.
- Use **nvprof** `--unified-memory-profiling off` to disable unified memory profiling
- Launch **nvvp**, Create New Session, and be sure to check the option for "Enable unified memory profiling".
- When viewing the execution profile in **nvvp**, be sure to use the "zoom in" feature to get better view of the action.

Multi-GPU Applications: NCCL

- The Unified Memory system has many benefits for CUDA programers and their applications but multi-GPU programing doesn't exactly standout in this regard.
- Today it is not uncommon for servers to contain 8 or more GPU devices. In principle, scaling an application from one to many GPUs should provide a significant performance boost. In practice, the benefits of multi-GPU development can be difficult to obtain – typically for three (correlated) reasons:
  - Poor exposure of parallelism in the problem
  - Devices exchange too much data and spend more time communicating than computing
  - PCI-e bandwidth limitations
- NCCL (pronounced "Nickel") is an open-source library from NVIDIA containing multi-GPU collective communication primitives in the spirit of MPI that are topology-aware and can be "easily" integrated into CUDA applications.
- NCCL with NVLINK creates a truly unique multi-GPU development opportunity.

The Future of Unified Memory

- On Kepler and Maxwell architectures, bulk managed data migration to the device is triggered by a kernel launch.
- Pages of device memory are allocated **before** they are used. This means it is not possible to oversubscribe device memory.

```
1  void foo() {
2
3    // init pointer to data
4    char *data;
5
6    // specify 32 GB of memory in bytes
7    size_t numBytes = 32*1024*1024*1024;
8
9    // Allocate 32 GB
10   cudaMallocManaged(&data, numBytes);
11 }
```

Listing 1: Fails on Kepler and Maxwell

The Future: GPU Page Faults

- On new Pascal architecture, managed memory uses GPU page faults rathe than bulk data migration. Pages of device memory are populated and data migrated on first touch.
- When kernel launches and accesses memory, if device does not have a VA translation, it issues an interrupt to CPU.
- The Unified Memory driver could decide to map the associated memory or migrate depending on various heuristics.
- With Pascal *on-demand paging* it is possible to allocate much more memory than is physically available on the device.
- Furthermore, the Pascal architecture provides more support for concurrent access to memory from CPU and GPU (page-level coherency).
- As an example, with very large graph processing, it can be difficult to predict which edges and vertices will need to be resident on the device. With Pascal, bring only necessary vertices/edges on-demand allowing work on very large graphs that cannot fit into GPU memory.

The Future: Unified Memory Hints

- In general want to minimize GPU page faulting. Fault handling can take 10s of μs while execution stalls.
- Clearly, want to keep data local to the accessing processor to minimize latency and maximize bandwidth. However, it is not difficult to think of situations which could induce "thrashing" whereby the data migration overhead exceeds locality benefits.
- The CUDA 8.0 runtime API will include unified memory "hints" such as `cudaMemPrefetchAsync` and `cudaMemAdvise` which allow for specification of allocation and usage policies for memory regions and manual prefetching.
- For example, using `cudaMemPrefetchAsync` can prefetch large array of data to avoid excessive number of expensive page faults on kernel launch.

Source Code Examples

CUDA Malloc/Memcpy Example


```
1   // number of bytes to alloc for arrays
2   size_t numBytes = N*sizeof(float);
3
4   // init host and device pointers
5   float *ha, *da;
6
7   // alloc host memory/arrays (pagable memory)
8   ha = (float*)malloc(numBytes);
9
10  // mem alloc arrays on the GPU device
11  cudaMalloc(&da,numBytes);
12
13  // copy host array to device array (synchronous)
14  cudaMemcpy(da, ha, numBytes, cudaMemcpyHostToDevice);
15
16  // launch kernels etc ... (asynchronous)
17
18  // now copy device array to host array (snychronous)
19  cudaMemcpy(ha, da, numBytes, cudaMemcpyDeviceToHost);
20
21  // dealloc on host and device
22  free(ha);  cudaFree(da);
```

Listing 2: The classic CUDA memory initialization strategy.

CUDA Zero-Copy Example

```
1   size_t numBytes = N*sizeof(float);
2
3   // init host and device pointers
4   float *ha, *da;
5
6   // alloc host memory (host pinned, device mapped)
7   cudaHostAlloc(&ha,numBytes,cudaHostAllocMapped);
8
9   // get device pointers for mapped memory
10  cudaHostGetDevicePointer(&da, ha,0);
11
12  // launch kernels etc ...
13  mykernel<<<blksz, grdsz>>>(da);
14
15  // wait for kernel to finish
16  cudaDeviceSynchronize();
17
18  // free host pointers
19  cudaFreeHost(ha);
```

Listing 3: Using cudaHostAlloc allows CUDA kernels to read and write those memory addresses. Note that kernel launch must reference device pointer. Only use zero-copy strategy with small amounts of data since the data is not cached on the GPU.

CUDA Zero-Copy with UVA Example


```
1   // number of bytes to alloc for arrays
2   size_t numBytes = N*sizeof(float);
3
4   // init host pointer
5   float *ha;
6
7   // alloc host memory (host pinned, device mapped)
8   cudaHostAlloc(&ha,numBytes,cudaHostAllocMapped);
9
10  // launch kernels etc ...
11  mykernel<<<blksz, grdsz>>>(ha);
12
13  // wait for kernel to finish
14  cudaDeviceSynchronize();
15
16  // free host pointers
17  cudaFreeHost(ha);
```

Listing 4: Using cudaHostAlloc allows CUDA kernels to read and write host memory addresses. With UVA in effect (introduced in CUDA 4.0), host pointers can be directly passed to kernel launch. Again, only use zero-copy strategy with small amounts of data since the data is not cached on the GPU.

CUDA Managed Memory Example


```
1   // number of bytes to alloc for arrays
2   size_t numBytes = N*sizeof(float);
3
4   // init host pointer
5   float *ha;
6
7   // alloc fully managed host memory
8   cudaMallocManaged(&ha,numBytes);
9
10  // launch kernels etc ...
11  mykernel<<<blksz, grdsz>>>(ha);
12
13  // wait for kernel to finish
14  cudaDeviceSynchronize();
15
16  // free host pointers
17  cudaFreeHost(ha);
```

Listing 5: Using cudaMallocManaged (introduced in CUDA 6.0) eliminates all explicit memory copies. Thanks to UVA, host pointers can be passed directly to kernal launch. Unlike zero-copy, using managed memory is generally quite scalable since the driver will automatically migrate the necessary data back and forth between host and device.

Unified Memory: Exclusive Access

```
1
2   // static allocation of manage memory in global scope
3   __device__ __managed__ int x, y=2;
4
5   // define generic kernel
6   __global__ void kernel() { x = 10; }
7
8   int main(void) {
9
10      // launch kernel (asynchronous)
11      kernel<<<1,1>>>();
12
13      // access managed memory while kernel executing ...
14      y = 20; // ERROR -- GPU has exclusive access
15
16      // wait for kernel to finish
17      cudaDeviceSynchronize();
18
19      return 0;
20  }
```

Listing 6: Do not do this. The device has exclusive access to managed memory while kernel launch is active, regardless of whether the kernel references the memory accessed by the host.

GPUDirect P2P UVA Memory Example

```
1   // check if each device can support peer access
2   cudaDeviceCanAccessPeer(&peerAccess01, 0, 1);
3   cudaDeviceCanAccessPeer(&peerAccess10, 1, 0);
4
5   // enable peer access: device 0 to device 1
6   cudaSetDevice(0);
7   cudaDeviceEnablePeerAccess(1,cudaPeerAccessDefault);
8
9   // enable peer access: device 1 to device 0
10  cudaSetDevice(1);
11  cudaDeviceEnablePeerAccess(0,cudaPeerAccessDefault);
12
13  // UVA enabled copy, memcpy knows who owns the pointers
14  cudaMemcpy(buf0, buf1, numBytes, cudaMemcpyDefault);
```

Listing 7: Exchanging memory between peer devices with UVA in effect. What makes P2P exchange special is that devices exchange data directly, effectively bypassing the host. This feature is often referred to as "GPUDirect P2P Transfer". Without P2P capabilities enabled there is an extra copy involved over PCI-e as data from the source device must be copied to the host and then copied again from the host to the destination.