CUDA 2.2 Pinned Memory APIs

July 2012

Table of Contents

Table of Contents **1**

1. Overview **2**

1.1 "Portable pinned memory": available to all contexts 3

1.2 "Mapped pinned memory": zero-copy 3

1.3 Write-combined memory 4

2 Driver API **5**

2.1 New device attributes 5

2.2 cuCtxCreate 5

2.3 cuMemHostAlloc 6

2.4 cuMemHostGetDevicePointer 6

3 CUDA Runtime API **7**

3.1 New Device Properties 7

3.2 cudaSetDeviceFlags 7

3.3 cudaHostAlloc 8

3.3 cudaHostGetDevicePointer 8

4 Frequently Asked Questions **9**

4.1 I am trying to use mapped pinned memory, but I'm not getting a device pointer 9

4.2 Why didn't NVIDIA implement zero-copy simply by ignoring the copy commands? 9

4.3 When should I use mapped pinned memory? 9

4.4 I am trying to use mapped pinned memory, and I'm not getting the expected results. 10

4.5 Why do pinned allocations seem to be using CUDA address space? 10

4.6 Mapped pinned memory is giving me a big performance hit! 11

4.7 When should I use write-combined memory? 11

1. Overview

The term "pinned memory" does not appear anywhere in the CUDA header files, but has been adopted by the CUDA developer community to refer to memory allocated by the CUDA driver API's ***cuMemAllocHost()*** or the CUDA runtime's ***cudaMallocHost()*** functions. Such memory is allocated for the CPU, but also page-locked and mapped for access by the GPU for higher transfer speeds and eligibility for asynchronous memcpy[1].

However, before CUDA 2.2 the benefits of pinned memory were realized only on the CPU thread (or, if using the driver API, the CUDA context) in which the memory was allocated. This restriction is especially problematic on pre-CUDA 2.2 applications that are operating multiple GPUs, since a given buffer was guaranteed to be treated as pageable by one of the CUDA contexts needed to drive multiple GPUs.

In addition, before CUDA 2.2, pinned memory could only be copied to and from a GPU's device memory; CUDA kernels could not access CPU memory directly, even if it was pinned.

CUDA 2.2 introduces new APIs that relax these restrictions via a new function called ***cuMemHostAlloc()***[2] (or in the CUDA runtime, ***cudaHostAlloc()***). The new features are as follows:

- "Portable" pinned buffers that are available to all GPUs.
- "Mapped" pinned buffers that are mapped into the CUDA address space. On integrated GPUs, mapped pinned memory enables applications to avoid superfluous copies since integrated GPUs operate on the same pool of physical memory as the CPU. As a result, mapped pinned buffers may be referred to as "zero-copy" buffers.
- "WC" (write-combined) memory that is not cached by the CPU, but kept in a small intermediary buffer and written as needed at high speed. WC memory has higher PCI Express copy performance and does not have any effect on the CPU caches (since the WC buffers are a separate hardware resource), but WC memory has drawbacks. The CPU cannot read from WC memory without incurring a performance penalty[3], so WC memory cannot be used in the general case – it is best for buffers where the CPU is

[1] Pageable memory cannot be copied asynchronously since the operating system may move it or swap it out to disk before the GPU is finished using it.

[2] You may wonder why NVIDIA would name a function "cuMemHostAlloc" when the existing function to allocate pinned memory is called "cuMemAllocHost." Both naming conventions follow "global to local" scoping as you read from left to right –prefix, function family, action. cuMemAllocHost() belongs to the family "Mem" and performs the "alloc host" operation; cuMemHostAlloc() belongs to the family "MemHost" and performs the "alloc" function.

[3] Note, SSE4.1 introduced the MOVNTDQA instruction that enables CPUs to read from WC memory with high performance.

producing data for consumption by the GPU. Additionally, WC memory may require fence instructions to ensure coherence.[4]

These features are completely orthogonal - you can allocate a portable, write-combined buffer, a portable pinned buffer, a write-combined buffer that is neither portable nor pinned, or any other permutation enabled by the flags.

1.1 "Portable pinned memory": available to all contexts

Before CUDA 2.2, the benefits of pinned memory could only be realized on the CUDA context that allocated it. This restriction was especially onerous on multi-GPU applications, since they often divide problems among GPUs, dispatching different subsets of the input data to different GPUs and gathering the output data into one buffer. ***cuMemHostAlloc ()*** relaxes this restriction through the `CU_MEMALLOC_PORTABLE` flag. When this flag is specified, the pinned memory is made available to all CUDA contexts, not just the one that performed the allocation[5].

Portable pinned memory works both for contexts that predate the allocation, and for contexts that are created after the allocation has been performed.

Portable pinned memory may be freed by any CUDA context by calling **cuMemFreeHost()**. Once freed, it is no longer available to any CUDA context.

The CUDA runtime exposes this feature via the new ***cudaHostAlloc()*** function with the `cudaHostAllocPortable` flag. Memory allocated by **cudaHostAlloc()** may be freed by calling **cudaFreeHost()**.

1.2 "Mapped pinned memory": zero-copy

To date, CUDA has presented a memory model where the CPU and GPU have distinct memory that is accessible to one device or the other, but never both. Data interchange between the two devices is achieved by allocating two buffers (one each in CPU memory and GPU memory) and copying data between them. This memory model reflects the target GPUs for CUDA, which historically have been discrete GPUs with dedicated memory subsystems.

There are two scenarios where it is desirable for the CPU and GPU to share a buffer without explicit buffer allocations and copies:

1) On GPUs integrated into the motherboard, the copies are superfluous because the CPU and GPU memory are physically the same.

[4] The CUDA driver uses WC internally and must issue a store fence instruction whenever it sends a command to the GPU. So the application may not have to use store fences at all.

[5] Portable pinned memory is not the default for compatibility reasons. Making pinned allocations portable without an opt-in from the application could cause failures to be reported that did not occur in previous versions of CUDA.

2) On discrete GPUs running workloads that are transfer-bound, or for suitable workloads where the GPU can overlap computation with kernel-originated PCI-Express transfers, higher performance may be achieved as well as obviating the need to allocate a GPU memory buffer.

Using this feature, NVIDIA has observed performance benefits that range from a modest 20% to more than 100%. There are also benefits in ease of programming as compared to streamed applications that overlap transfers with computation. CUDA streams require developers to create streams and software-pipeline downloads, kernel processing and readbacks; and the optimal number of streams must be determined empirically and may vary from chip to chip. In contrast, an application using zero-copy simply launches a kernel with device pointers that reference pinned CPU memory.

The API works by creating pinned allocations that have two addresses that alias the same memory: a host pointer that is passed back upon allocation, and a device pointer that may be queried. The buffer is freed using existing API functions that free pinned memory (the driver API's ***cuMemFreeHost()*** or the CUDA runtime's ***cudaFreeHost()***).

Since the mapped buffer is shared by the CPU and GPU, developers must synchronize access using existing context, stream or event APIs. Synchronization typically is needed when the application wants to read data written to a mapped pinned buffer by the GPU. If the application is using multi-buffering to maximize concurrent CPU/GPU execution, it may have to synchronize before starting to write to a buffer that may still be getting read by the GPU.

1.3 Write-combined memory

By default, CUDA allocates pinned memory as cacheable so it can serve as a substitute for memory allocated with traditional APIs such as ***malloc()***. But there are potential performance benefits to allocating pinned memory as write-combined (WC). Quoting from Intel's web site[6]: "Writes to WC memory are not cached in the typical sense of the word cached. They are delayed in an internal buffer that is separate from the internal L1 and L2 caches. The buffer is not snooped and thus does not provide data coherency."

So writes to write-combined memory do not pollute the CPU caches, and because the memory is not snooped during transfers across the PCI Express bus, it may perform as much as 40% faster on certain PCI Express 2.0 implementations.

Certain CPU instructions are especially designed to interact with WC memory – these include "nontemporal store" instructions such as SSE's MOVNTPS and "masked move" instructions such as SSE2's MASKMOVDQU. Using such instructions is not necessary to get the benefits of WC memory, however; any set of CPU instructions that write the data once, preferably contiguously, will exhibit good performance on WC memory.

The SSE4 instruction set provides a streaming load instruction (MOVNTDQA) that can efficiently read from WC memory[7]. If the CPUID instruction is executed with EAX==1, bit 19 of ECX indicates whether SSE4.1 is available.

[6] http://download.intel.com/design/PentiumII/applnots/24442201.pdf

[7] http://softwarecommunity.intel.com/articles/eng/1248.htm

2 Driver API

This section briefly describes the CUDA 2.2 additions to the CUDA driver API for pinned memory management.

2.1 New device attributes

```
CUresult  CUDAAPI cuDeviceGetAttribute(int *pi,
CUdevice_attribute attrib, CUdevice dev );
```

New enumerants `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` and `CU_DEVICE_ATTRIBUTE_INTEGRATED` that enable applications to query whether a device can map system memory and whether it is integrated, respectively.

`CU_DEVICE_ATTRIBUTE_INTEGRATED`	Nonzero if the device is integrated with the host memory system.
`CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY`	Nonzero if the device can map host memory. This structure member corresponds to the driver API's `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` query.

For integrated devices, zero-copy (specifying `CU_CTX_MAP_HOST` to ***cuCtxCreate()***, specifying `CU_MEMHOSTALLOC_DEVICEMAP` to ***cuMemHostAlloc()***, and using the resulting buffer for data interchange in conjunction with context-, stream- or event-based CPU synchronization) is always a performance win. All CUDA-capable integrated devices are able to map host memory.If the `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` query returns 0, the device cannot map system memory and ***cuMemHostGetDevicePointer()/ cudaHostGetDevicePointer()*** will return a failure code. On such devices, it is still legitimate to specify the `CU_MEMHOSTALLOC_DEVICEMAP/cudaHostAllocMapped` flag, because portable pinned allocations may make the memory available to other devices that are able to map host memory.

If the `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` query returns 1, the device pointer for a mapped pinned buffer may be queried with ***cuMemHostGetDevicePointer()/cudaHostGetDevicePointer()***.

2.2 cuCtxCreate

```
CUresult  CUDAAPI cuCtxCreate(CUcontext *pctx, unsigned int
flags, CUdevice dev );
```

specified to ***cuCtxCreate()*** to enable mapped pinned allocations. It may be combined with other flags such as `CU_CTX_SCHED_SPIN`.

If this flag is specified, pinned allocations may be mapped into the CUDA address space by specifying the `CU_MEMHOSTALLOC_DEVICEMAP` flag and their CUDA addresses (`CUdeviceptr`) may be queried by calling ***cuMemGetDevicePointer()***.

2.3 cuMemHostAlloc

```
CUresult cuMemHostAlloc( void **pp, size_t cBytes, unsigned
int Flags );
```

```
#define CU_MEMHOSTALLOC_PORTABLE        0x01
#define CU_MEMHOSTALLOC_DEVICEMAP       0x02
#define CU_MEMHOSTALLOC_WRITECOMBINED   0x04
```

This function allocates a pinned buffer that may be directly accessed by the GPU(s) in the system and passes back the resulting CPU pointer in `*pp`.

If the `Flags` parameter is 0, ***cuMemHostAlloc()*** emulates the pre-CUDA 2.2 function ***cuMemAllocHost()*** except that it may allocate more than 4GB of memory on 64-bit systems.

If `CU_MEMHOSTALLOC_PORTABLE` is specified, the memory is made available to all CUDA contexts, including contexts created after the memory has been allocated. Any context may free the memory by calling cuMemFreeHost(); at that time, the memory is freed for all contexts.

If `CU_MEMHOSTALLOC_DEVICEMAP` is specified, the memory is mapped into the CUDA address space. The device pointer may be obtained by calling ***cuMemHostGetDevicePointer ()***.

If `CU_MEMHOSTALLOC_WRITECOMBINED` is specified, the memory is allocated and mapped as write-combined.

The pointer allocated by ***cuMemHostAlloc()*** must be freed with the existing driver API function ***cuMemFreeHost()***.

2.4 cuMemHostGetDevicePointer

```
CUresult cuMemHostGetDevicePointer( CUdeviceptr *pDevice,
void *pHost, unsigned int Flags );
```

Passes back the device pointer for a host allocation that was allocated by ***cuMemHostAlloc()*** with the `CU_MEMHOSTALLOC_DEVICEMAP` flag set. The function will fail if either of these conditions is not met by the input host pointer.

IMPORTANT: for portable allocations, there is no guarantee that the device pointer will be the same for different GPUs. This function must be called on each CUDA context by applications that wish to perform zero-copy operations with multiple GPUs.

At present, `Flags` must be set to 0.

3 CUDA Runtime API

This section briefly describes the CUDA 2.2 additions to the CUDA runtime APIs for pinned memory management.

3.1 New Device Properties

The cudaDeviceProp structure passed back by ***cudaGetDeviceProperties()*** has two new members:

`int integrated;`	Nonzero if the device is integrated with the host memory system. This structure member corresponds to the driver API's `CU_DEVICE_ATTRIBUTE_INTEGRATED` query.
`int canMapHostMemory;`	Nonzero if the device can map host memory. This structure member corresponds to the driver API's `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` query.

3.2 cudaSetDeviceFlags

```
cudaError_t CUDARTAPI cudaSetDeviceFlags( unsigned int
Flags );
```

```
#define cudaDeviceScheduleAuto      0
#define cudaDeviceScheduleSpin      1
#define cudaDeviceScheduleYield     2
#define cudaDeviceBlockingSync      4
#define cudaDeviceMapHost           8
```

cudaSetDeviceFlags() may be called before any other CUDA operations are performed in a given CPU thread.

`cudaDeviceScheduleSpin` and `cudaDeviceScheduleYield` specify whether pageable memcpy's spin or yield the CPU for synchronization; `cudaDeviceScheduleAuto` allows CUDA to select the heuristic to use.

`cudaDeviceBlockingSync` specifies that the CPU will block rather than spin when ***cudaThreadSynchronize()*** is called.

The `cudaDeviceMapHost` flag causes all pinned allocations to consume CUDA address space, and enables ***cudaHostGetDevicePointer()*** to obtain the device pointer for a given host allocation. If ***cudaSetDeviceFlags()***was not called before CUDA started to be used, ***cudaHostGetDevicePointer()*** will return a failure code.

3.3 cudaHostAlloc

```
cudaError_t CUDARTAPI cudaHostAlloc( void **pHost, size_t
bytes, unsigned int Flags );
#define cudaHostAllocDefault        0
#define cudaHostAllocPortable       1
#define cudaHostAllocMapped         2
#define cudaHostAllocWriteCombined  4
```

This function allocates a pinned buffer that may be directly accessed by the GPU(s) in the system and passes back the resulting CPU pointer in `*pp`.

If the `Flags` parameter is 0 or `cudaHostAllocDefault`, ***cudaHostAlloc()*** emulates the pre-CUDA 2.2 function ***cudaMallocHost()*** except that it may allocate more than 4GB of memory on 64-bit systems.

If `cudaHostAllocPortable` is specified, the memory is made available to all CUDA contexts, including contexts created after the memory has been allocated.

If `cudaHostAllocMapped` is specified, the memory is mapped into the CUDA address space. The device pointer may be obtained by calling ***cuMemHostGetDevicePointer ()***.

If `cudaHostAllocWriteCombined` is specified, the memory is allocated and mapped as write-combined.

The pointer allocated by ***cudaHostAlloc()*** must be freed with the existing driver API function ***cudaFreeHost()***.

3.3 cudaHostGetDevicePointer

```
cudaError_t CUDARTAPI cudaHostGetDevicePointer( void
**pDevice, void *pHost, unsigned int Flags );
```

Passes back the device pointer for a host allocation that was allocated by ***cudaHostAlloc()*** with the `cudaHostAllocMapped` flag. The function will fail if either of these conditions is not met by the input host pointer.

IMPORTANT: for portable allocations, there is no guarantee that the device pointer will be the same for different GPUs. This function must be called for each CUDA device by applications that wish to perform zero-copy operations with multiple GPUs.

At present, Flags must be set to 0.

4 Frequently Asked Questions

This section is by no means complete, but addresses a few questions that are commonly encountered by developers trying to use the new CUDA 2.2 pinned memory features.

4.1 I am trying to use mapped pinned memory, but I'm not getting a device pointer.

If ***cuMemHostGetDevicePointer()*** or ***cudaMemHostGetDevicePointer()*** is failing, this is usually because the CUDA context was not created with the needed flags. Driver API apps must create the context with the `CU_CTX_MAP_HOST` flag; CUDA runtime apps must call ***cudaSetDeviceFlags()*** with the `cudaMapHost` flag before any CUDA operations have been performed[8].

Another possible explanation for ***cuMemHostGetDevicePointer()*** or ***cudaMemHostGetDevicePointer()***failing is that the GPU does not support mapped pinned memory. You can use the new device query described in section 2.1 and section 3.1 to determine whether mapped pinned memory is supported.

4.2 Why didn't NVIDIA implement zero-copy simply by ignoring the copy commands?

Although many CUDA applications exhibit an execution pattern (memcpy host→device, kernel processing, memcpy device→host) where this could be done, it would be very difficult to ignore the copy in the general case because once the host→device copy is done, the host source buffer can be modified by the CPU without affecting the data being processed by the GPU.

4.3 When should I use mapped pinned memory?

For integrated GPUs, mapped pinned memory is always a performance win because it eliminates superfluous memcpy's. You can check whether a given GPU is integrated with the driver API's ***cuDeviceGetAttribute()*** function with the CU_DEVICE_ATTRIBUTE_INTEGRATED enumeration value, or by calling the CUDA runtime's ***cudaGetDeviceProperties()*** function and checking the `integrated` member of the `cudaDeviceProp` structure.

[8] A future version of CUDA will be able to allocate mapped pinned memory on a per-allocation basis (based solely on whether the `CU_MEMHOSTALLOC_DEVICEMAP` flag or `cudaHostAllocMapped` flag was specified), without requiring a context-wide flag to be in effect.

For discrete GPUs, mapped pinned memory is only a performance win in certain cases. Since the memory is not cached by the GPU, it should be read or written exactly once; and the global loads and stores that read or write the memory must be coalesced to avoid a 2x performance penalty.

For most applications, this means that discrete GPUs must perform all coalesced memory transactions or zero-copy will not improve performance.

4.4 I am trying to use mapped pinned memory, and I'm not getting the expected results.

99% of the time, this is due to missing CPU/GPU synchronization. All kernel launches are asynchronous, i.e. control is returned to the CPU before the GPU has finished executing the kernel. Any kernel that writes to a mapped output buffer requires that the CPU synchronize with the GPU before reading from the buffer. In this sense, kernel launches that operate on mapped pinned memory are akin to asynchronous memcpy calls such as ***cuMemcpyDtoHAsync()*** or ***cudaMemcpyAsync()*** – since the GPU is operating independently of the CPU, the only way to ensure that the buffer is ready to be read is through explicit synchronization.

The big hammer to use for CPU/GPU synchronization is ***cuCtxSynchronize()/cudaThreadSynchronize()***, which wait until the GPU is idle before returning.

More granular CPU/GPU synchronization may be performed via streamed operations and calling ***cuStreamSynchronize()/cudaStreamSynchronize()***. These calls may return before the GPU goes idle, since other streams may contain work for the GPU.

The most operation-specific CPU/GPU synchronization may be performed via events – ***cuEventRecord()/cudaEventRecord()*** causes the GPU to record an event when all preceding GPU commands in the stream have been performed; ***cuEventSynchronize()/cudaEventSynchronize()*** wait until the event has been recorded.

4.5 Why do pinned allocations seem to be using CUDA address space?

Unfortunately, if you specify `CU_CTX_MAP_HOST` to ***cuCtxCreate()*** or specify `cudaDeviceMapHost` to ***cudaSetDeviceFlags()***, all pinned allocations are mapped into CUDA's 32-bit linear address space, regardless of whether the device pointer is needed. For some applications, this may limit the usefulness of mapped pinned memory to prototyping and proof-of-concept implementations.

A future version of CUDA will obviate the need for these context- and device-wide flags to be specified and enable allocations to be mapped on a per-allocation basis instead.

4.6 Mapped pinned memory is giving me a big performance hit!

On discrete GPUs, mapped pinned memory is only a performance win if all memory transactions are coalesced. For uncoalesced memory transactions, a 2-7x reduction in PCI Express bandwidth will be observed.

4.7 When should I use write-combined memory?

Since it is inefficient for the CPU to read from WC memory, it cannot be used in the general case. Developers must use *a priori* knowledge of the buffer's usage in deciding whether to allocate them as WC. If the application never uses the CPU to read from the buffer (except via SSE4's MOVNTDQA instruction), the buffer is a good candidate for WC allocation.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com