

Pinned Memory

Allocated host memory is by default pageable, that is, subject to page fault operations that move data in host virtual memory to different physical locations as directed by the operating system. Virtual memory offers the illusion of much more main memory than is physically available, just as the L1 cache offers the illusion of much more on-chip memory than is physically available.

The GPU cannot safely access data in pageable host memory because it has no control over when the host operating system may choose to physically move that data. When transferring data from pageable host memory to device memory, the CUDA driver first allocates temporary page-locked or pinned host memory, copies the source host data to pinned memory, and then transfers the data from pinned memory to device memory. The CUDA runtime allows you to directly allocate pinned host memory using:

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

This function allocates count bytes of host memory that is page-locked and accessible to the device. Since the pinned memory can be accessed directly by the device, it can be read and written with much higher bandwidth than pageable memory. However, allocating excessive amounts of pinned memory might degrade host system performance, since it reduces the amount of pageable memory available to the host system for storing virtual memory data. The Pinned host memory must be freed with `cudaError_t cudaFreeHost(void *ptr);`

Zero-Copy Memory

In general, the host cannot directly access device variables, and the device cannot directly access host variables. There is one exception to this rule: zero-copy memory. Both the host and device can access zero-copy memory. GPU threads can directly access zero-copy memory. There are several advantages to using zero-copy memory in CUDA kernels, such as:

- Leveraging host memory when there is insufficient device memory
- Avoiding explicit data transfer between the host and device
- Improving PCIe transfer rates

When using zero-copy memory to share data between the host and device, you must synchronize memory accesses across the host and device. Modifying data in zero-copy memory from both the host and device at the same time will result in undefined behavior

Zero-copy memory is pinned (non-pageable) memory that is mapped into the device address space. You can create a mapped, pinned memory region with the following function:

```
cudaError_t cudaHostAlloc(void **pHost, size_t count, unsigned int flags);
```

This function allocates count bytes of host memory that is page-locked and accessible to the device. Memory allocated by this function must be freed with `cudaFreeHost`.

The flags parameter enables further configuration of special properties of the allocated memory:

- `cudaHostAllocDefault`
- `cudaHostAllocPortable`
- `cudaHostAllocWriteCombined`
- `cudaHostAllocMapped`

`cudaHostAllocDefault` makes the behavior of `cudaHostAlloc` identical to `cudaMallocHost`. Setting `cudaHostAllocPortable` returns pinned memory that can be used by all CUDA contexts, not just the one that performed the allocation. The flag `cudaHostAllocWriteCombined` returns write-combined memory, which can be transferred across the PCI Express bus more quickly on some system configurations but cannot be read efficiently by most hosts.

Therefore, write-combined memory is a good option for buffers that will be written by the host and read by the device using either mapped pinned memory or host-to-device transfers. The most relevant flag to zero-copy memory is `cudaHostAllocMapped`, which returns host memory that is mapped into the device address space. You can obtain the device pointer for mapped pinned memory using the following function:

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

Using zero-copy memory as a supplement to device memory with frequent read/write operations will significantly slow performance. Because every memory transaction to mapped memory must pass over the PCIe bus, a significant amount of latency is added even when compared to global memory. You This function returns a device pointer in `pDevice` that can be referenced on the device to access mapped, pinned host memory. This function will fail if the device does not support mapped, pinned memory. `flag` is reserved for future use. For now, it must be set to zero.

Unified Virtual Addressing

Devices with compute capability 2.0 and later support a special addressing mode called Unified Virtual Addressing (UVA). UVA, introduced in CUDA 4.0, is supported on 64-bit Linux systems. With UVA, host memory and device memory share a single virtual address space. Prior to UVA, you needed to manage which pointers referred to host memory and which referred to device memory. Using UVA, the memory space referenced by a pointer becomes transparent to application code. Under UVA, pinned host memory allocated with `cudaHostAlloc` has identical host and device pointers. You can therefore pass the returned pointer directly to a kernel function. Recall the zero-copy example from the previous section where you:

- Allocated mapped, pinned host memory
- Acquired the device pointer to the mapped, pinned memory using a CUDA runtime function
- Passed the device pointer to your kernel.

With UVA, there is no need to acquire the device pointer or manage two pointers to what is physically the same data.

Unified Memory

With CUDA 6.0, a new feature called Unified Memory was introduced to simplify memory management in the CUDA programming model. Unified Memory creates a pool of managed memory, where each allocation from this memory pool is accessible on both the CPU and GPU with the same memory address (that is, pointer). The underlying system automatically migrates data in the unified memory space between the host and device. This data movement is transparent to the application, greatly simplifying the application code. Unified Memory depends on Unified Virtual Addressing (UVA) support, but they are entirely different technologies. UVA provides a single virtual memory address space for all processors in the system. However, UVA does not automatically migrate data from one physical location to another; that is a capability unique to Unified Memory. Unified Memory offers a “single-pointer-to-data” model that is conceptually similar to zero-copy memory. However, zero-copy memory is allocated in host memory, and as a result kernel performance generally suffers from high latency accesses to zero-copy memory over the PCIe bus. Unified Memory, on the other hand, decouples memory and execution spaces so that data can be transparently migrated on demand to the host or device to improve locality and performance. Managed memory refers to Unified Memory allocations that are automatically managed by the underlying system and is interoperable with device-specific allocations, such as those created using the `cudaMalloc` routine. Therefore, you can use both types of memory in a kernel: managed memory that is controlled by the system, and un-managed memory that must be explicitly allocated and transferred by the application. All CUDA operations that are valid on device memory are also valid on managed memory. The primary difference is that the host is also able to reference and access managed memory. Managed memory can be allocated statically or dynamically. You can statically declare a device variable as a

managed variable by adding a `__managed__` annotation to its declaration. This can only be done in file-scope and global-scope. The variable can be referenced directly from either host or device code:

```
__device__ __managed__ int y;
```

You can also allocate managed memory dynamically using the following CUDA runtime function:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);
```

This function allocates `size` bytes of managed memory and returns a pointer in `devPtr`. The pointer is valid on all devices and the host. The behavior of a program with managed memory is functionally unchanged to its counterpart with un-managed memory. However, a program that uses managed memory can take advantage of automatic data migration and duplicate pointer elimination. In CUDA 6.0, device code cannot call `cudaMallocManaged`. All managed memory must be dynamically allocated from the host or statically declared in global scope.

MEMORY ACCESS PATTERNS

Most device data access begins in global memory, and most GPU applications tend to be limited by memory bandwidth. Therefore, maximizing your application's use of global memory bandwidth is a fundamental step in kernel performance tuning. If you do not tune global memory usage properly, other optimizations will likely have a negligible effect.

To achieve the best performance when reading and writing data, memory access operations must meet certain conditions. One of the distinguishing features of the CUDA execution model is that instructions are issued and executed per warp. Memory operations are also issued per warp. When executing a memory instruction, each thread in a warp provides a memory address it is loading or storing. Cooperatively, the 32 threads in a warp present a single memory access request comprised of the requested addresses, which is serviced by one or more device memory transactions. Depending on the distribution of memory addresses within a warp, memory accesses can be classified into different patterns. In this section, you are going to examine different memory access patterns and learn how to achieve optimal global memory access.

Aligned and Coalesced Access

Global memory loads/stores are staged through caches. Global memory is a logical memory space that you can access from your kernel. All application data initially resides in DRAM, the physical device memory. Kernel memory requests are typically served between the device DRAM and SM on-chip memory using either 128-byte or 32-byte memory transactions.

All accesses to global memory go through the L2 cache. Many accesses also pass through the L1 cache, depending on the type of access and your GPU's architecture. If both L1 and L2 caches are used, a memory access is serviced by a 128-byte memory transaction. If only the L2 cache is used, a memory access is serviced by a 32-byte memory transaction. On architectures that allow the L1 cache to be used for global memory caching, the L1 cache can be explicitly enabled or disabled at compile time. An L1 cache line is 128 bytes, and it maps to a 128-byte aligned segment in device memory. If each thread in a warp requests one 4-byte value, that results in 128 bytes of data per request, which maps perfectly to the cache line size and device memory segment size.

There are two characteristics of device memory accesses that you should strive for when optimizing your application:

- Aligned memory accesses
- Coalesced memory accesses

Aligned memory accesses occur when the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction (either 32 bytes for L2 cache or 128 bytes for L1 cache).

Performing a misaligned load will cause wasted bandwidth. Coalesced memory accesses occur when all 32 threads in a warp access a contiguous chunk of memory. Aligned coalesced memory accesses are ideal: A warp accessing a contiguous chunk of memory starting at an aligned memory address. To maximize global memory throughput, it is important to organize memory operations to be both aligned and coalesced.

Global Memory Reads

In an SM, data is pipelined through one of the following three cache/buffer paths, depending on what type of device memory is being referenced:

- L1/L2 cache
- Constant cache
- Read-only cache

L1/L2 cache is the default path. To pass data through the other two paths requires explicit management by the application but can lead to performance improvement depending on the access patterns used. Whether global memory load operations pass through the L1 cache depends on two factors:

- Device compute capability
- Compiler options

On Fermi GPUs (compute capability 2.x) and Kepler K40 or later GPUs (compute capability 3.5 and up), L1 caching of global memory loads can be either enabled or disabled with compiler flags. By default, the L1 cache is enabled for global memory loads on Fermi devices and disabled on K40 and later GPUs. The following flags inform the compiler to disable the L1 cache `-Xptxas -dlcm=cg`. With the L1 cache disabled, all load requests to global memory go directly to the L2 cache; when an L2 miss occurs, the requests are serviced by DRAM. Each memory transaction may be conducted by one, two, or four segments, where one segment is 32 bytes. The L1 cache can also be explicitly enabled with the following flag `-Xptxas -dlcm=ca`. With this flag set, global memory load requests first attempt to hit in L1 cache. On an L1 miss, the requests go to L2. On an L2 miss, the requests are serviced by DRAM. In this mode, a load memory request is serviced by a 128-byte device memory transaction.

Cached Loads

Cached load operations pass through L1 cache and are serviced by device memory transactions at the granularity of an L1 cache line, 128-bytes. Cached loads can be classified as aligned/misaligned and coalesced/uncoalesced. The ideal case is aligned and coalesced memory accesses. The addresses requested by all threads in a warp fall within one cache line of 128 bytes. Only a single 128-byte transaction is required to complete the memory load operation. Bus utilization is 100 percent, and there is no unused data in this transaction.

Global Memory Writes

Memory store operations are relatively simple. The L1 cache is not used for store operations on either Fermi or Kepler GPUs, store operations are only cached in the L2 cache before being sent to device memory. Stores are performed at a 32-byte segment granularity. Memory transactions can be one, two, or four segments at a time. For example, if two addresses fall within the same 128-byte region but not within an aligned 64-byte region, one four-segment transaction will be issued (that is, issuing a single four-segment transaction performs better than issuing two one-segment transactions).