Dr.Dobb's THE WORLD OF SOFTWARE DEVELOPMENT

CUDA, Supercomputing for the Masses: Part 11

Revisiting CUDA memory spaces

March 18, 2009
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921

In CUDA, Supercomputing for the Masses: Part 10 of this article series on CUDA (short for "Compute Unified Device Architecture"), I examined CUDPP, the "CUDA Data Parallel Primitives Library." In this installment, I revisit local and constant memory and introduce the concept of "texture memory."

Optimizing the performance of CUDA applications most often involves optimizing data accesses which includes the appropriate use of the various CUDA memory spaces. Texture memory provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global, and shared memory) and dedicated interpolation hardware separate from the thread processors. Texture memory also provides a way to interact with the display capabilities of the GPU. Texture memory is an extensive and evolving topic that will be introduced here and discussed more in a dedicated follow-on article.. Part 4 of this series introduced the CUDA memory model and illustrated the various CUDA memory spaces with the schematic in Figure 1. Each of these memory spaces has certain performance characteristics and restrictions.

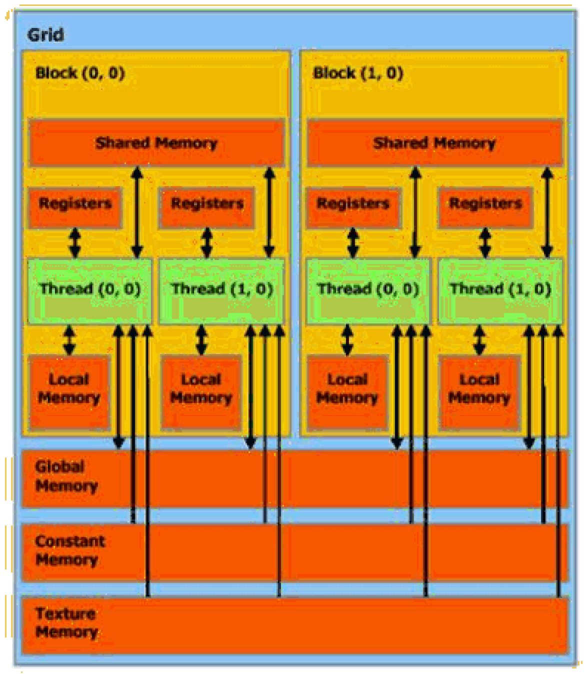

Figure 1: CUDA memory spaces.

Appropriate use of these memory spaces can have significant performance implications for CUDA applications. Table 1 summarizes characteristics of the various CUDA memory spaces. Part 5 of this series discusses CUDA memory spaces (with the exception of texture memory) in greater detail and includes performance cautions.

Memory	Location	Cached	Access	Scope
Register	On-chip	No	Read/write	One thread
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read (CUDA 2.1 and previous)	All threads + host

Table 1: Characteristics of the various CUDA memory spaces.

Local Memory

Local memory is a memory abstraction that implies "local in the scope of each thread". It is not an actual hardware component of the multi-processor. In actuality, local memory resides in global memory allocated by the compiler and delivers the same performance as any other global memory region. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. Unfortunately, the relationship between automatic variables and local memory continues to be a source of confusion for CUDA programmers. The compiler might choose to place automatic variables in local memory when:

- There are too many register variables.
- A structure would consume too much register space.
- The compiler cannot determine if an array is indexed with constant quantities. Please note that registers are not addressable so an array has to go into local memory -- even if it is a two-element array -- when the addressing of the array is not known at compile time.

For additional discussion, please see "Register/Local Memory Cautions" in Part 5 of this series, "The CUDA Memory Model" in Part 4, and the CUDA programming guide on the NVIDIA CUDA Zone Documentation site.

Constant Memory

Constant memory is read only from kernels and is hardware optimized for the case when all threads read the same location. Amazingly, constant memory provides one cycle of latency when there is a cache hit even though constant memory resides in device memory (DRAM). If threads read from multiple locations, the accesses are serialized. The constant cache is written to only by the host (not the device because it is constant!) with `cudaMemcpyToSymbol` and is persistent across kernel calls within the same application. Up to 64KB of data can be placed in constant cache and there is 8 KB of cache for each multiprocessor. Access to data in constant memory can range from one cycle for in cache data to hundreds of cycles depending on cache locality. The first access to constant memory often does not generate a cache "miss" due to pre-fetching.

Texture Memory in CUDA

Graphics processors provide texture memory to accelerate frequently performed operations such as mapping, or deforming, a 2D "skin" onto a 3D polygonal model. Tricks like those in Figure 2 let low-resolution game objects appear as visually richer objects with greater complexity and detail. To remain competitive, graphics card manufacturers added the extra hardware (e.g., texture units) to make these graphics operations occur very fast. CUDA provides mechanisms so C-programmers can exploit some of the added capabilities of the texture unit hardware on CUDA-enabled devices. (For more information on the use of textures in graphics, please see the free online GPU Gems books. A good place to start is Chapter 37 of GPU Gems 2.)

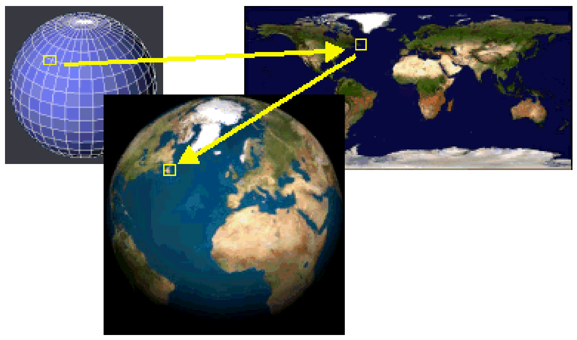

Figure 2: Texture memory in action.

The easiest way to think of texture memory is as an alternative memory access path that the CUDA programmer can bind to regions of the GPU device memory (e.g., global memory). Texture references can be bound to the same, overlapping or different textures in memory. Each on-chip texture unit has some internal memory that buffers data from global memory. For this reason, texture memory can be used as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements discussed in previous articles do not apply to texture memory accesses. This is particularly useful when targeting previous generation CUDA-capable GPUs but may not matter as much with the relaxed coalescing requirements in newer hardware.

Since optimized data access is very important to GPU performance, the use of texture memory can (in the right circumstances) provide a large performance increase. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. CUDA provides 1D, 2D and 3D fetch capabilities using texture memory. Since a texture performs a read operation from global memory only when there is a cache miss it is possible to exceed the maximum theoretical memory bandwidth of the underlying global memory through the judicious use of the texture memory cache. This makes the texture fetch (`texfetch`) rate the more useful metric for analyzing kernel performance when using textures. For example, Patrick LeGresley notes in slide 32 of High Performance Computing with CUDA that the G80 architecture can provide approximately 18 billion fetches per second.

Especially consider using textures when:

- There is locality of reference so caching in texture memory can help.
- Use of the texture cache can reduce the penalty for nearly coalesced accesses (such as a misaligned starting address) that cannot be made fully coalesced.

Each of the previous is problem dependent so some testing is required. Some authors have reported certain texture memory access patterns are many times faster than others (For one example, see Kipton Barro's presentation CUDA Tricks and Computational Physics).

Other performance benefits of texture memory (over global and constant memory) include:

- Packed data may be broadcast to separate variables in a single operation.
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] by the texture unit hardware.
- Linear, bilinear, and tri-linear interpolation using dedicated hardware separate from the thread processors.

Texture memory provides many more capabilities beyond those mentioned in this first introduction. A follow-on article will discuss texture memory further. Until the next article, please look to the NVIDIA_CUDA_SDK projects folder for examples. The Internet also contains many more useful examples as well that you can download and try. Following are two possibilities:

- A potentially helpful CUDA 3D texture example is located in the cookbook examples at the Google cudaiap2009 "cuda@mit" site.
- The CIRL fuzzy logic tutorial.

An excellent resource to learn more about the texture cache and other methods for data reuse on GPUs is Mark Silberstein's paper Efficient Computation of Sum-products on GPUs Through Software Managed Cache.

Summary

All of the above (and the previous articles in this series) indicate that CUDA developers must have a firm grasp of CUDA memory spaces. Specifically, be aware that local and global memory are not cached and their access latencies are high. Access latencies as reported by David Kirk and Wen-mei Hwu in The CUDA Memory Model with additional API and Tools info are:

- Register - dedicated HW - single cycle
- Shared Memory - dedicated HW - single cycle
- Local Memory - DRAM, no cache - *slow*
- Global Memory - DRAM, no cache - *slow*
- Constant Memory - DRAM, cached, one to hundreds of cycles depending on cache locality
- Texture Memory - DRAM, cached, hundreds of cycles
- Instruction Memory (invisible) - DRAM, cached

Consider a typical CUDA template as:

- Split a task into subtasks
- Divide input data into chunks that fit into registers and shared memory
- Load a data chunk from global memory into registers and shared memory
- Each data chunk is processed by a thread block
- Copy results back to global memory

However texture and constant memory are cached and can significantly increase application performance -- due to their access characteristics -- depending on the application access patterns:

- R/O no structure → constant memory
- R/O array structured → texture memory (CUDA 2.1 and previous)
- R/W shared within block → shared memory
- R/W registers spill to local memory and may provide a surprise slowdown
- R/W inputs/results → global memory

For More Information

- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.