CUDA, Supercomputing for the Masses: Part 4

Understanding and using shared memory (1)

June 03, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/208401741

One of the most important performance challenges facing CUDA (short for "Compute Unified Device Architecture") developers is the best use of local multiprocessor memory resources such as shared memory, constant memory, and registers. The reason discussed in Part 3 of this series is that while global memory can deliver over 60GB/s, this would translate to only 15GF/s for single-touch use of data -- getting higher performance requires local data reuse. The CUDA software and hardware designers have done some wonderful work to hide global memory latency and global memory bandwidth restrictions -- so long as there is some local data reuse.

Recall from Part 2 that a kernel launch requires the specification of an execution configuration to define the number of threads that compose a block and the number of blocks that are combined together to form a grid. It is important to note that threads within a block can communicate with each other through local multi-processor resources because the CUDA execution model specifies that a block can only be processed on a single multi-processor. In other words, data written to shared memory within a block is accessible to all other threads within that block, but it is not accessible to a thread from any other block. Shared memory with these characteristics can be implemented very efficiently in hardware which translates to fast memory accesses (with some caveats discussed shortly) for CUDA developers.

Now we have a way for the CUDA-enabled hardware designers to balance price versus the needs of the CUDA software developers. As developers, we want large amounts of local multiprocessor resources such as registers and shared memory. It makes our jobs much easier and our software more efficient. The hardware designer, on the other hand, needs to deliver hardware at a low price-point and unfortunately fast local multi-processor memory is expensive. We all agree that inexpensive CUDA hardware is wonderful, so CUDA-enabled hardware is designed to be marketed at various price-points with different capabilities. The market then decides on the appropriate price versus capability trade-offs. This is actually a very good solution because the technology is evolving quickly -- each new generation of CUDA-enabled devices is more powerful than the previous generation and contains ever greater numbers of higher performance components at the same price points of the previous generation.

Wait! This sounds more like a software headache than a compromise because the CUDA developer needs to account for all these different hardware configurations and we are challenged with limited amounts of device resources. To help, several design aids have been created to help select the "best" high-performance execution configurations for different architectures. I highly recommend downloading and playing with the CUDA occupancy calculator, which is simply a nicely done spreadsheet. (The nvcc compiler will report information for each kernel that is needed for the spreadsheet when passed the `--ptxas-options=-v` option such as the number of registers as well as local, shared, and constant memory usage.) Still, a common piece of advice in both the forums and documentation is, "try some different configurations and measure the effect on performance". This is easy to do since the execution configuration is specified by variables. In fact, many applications might be able to effectively auto configure themselves (e.g., determine the best execution configuration) when installed. Also, the CUDA runtime calls `cudaGetDeviceCount()` and `cudaGetDeviceProperties()` provide a way to enumerate the CUDA devices in a system and retrieve their properties. One possible way to use this information is to perform a table lookup for the best performing execution configurations or to jump start an auto tuner.

The CUDA Execution Model

To potentially increase performance, each hardware multiprocessor has the ability to actively process multiple blocks at one time. How many depends on the number of registers per thread and how much shared memory per block is required by a given kernel. The blocks that are processed by one multiprocessor at one time are referred to as active. Kernels with minimal resource requirements can better utilize (or occupy) each multiprocessor because the registers and shared memory of the multiprocessor are split among all the threads of the active blocks. Use the CUDA occupancy calculator to explore the trade-offs between number of threads and active blocks versus the number of registers and amount of shared memory. Finding the right combination can greatly increase the performance of your kernels. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. (See the Part 3 discussion on `cudaGetLastError()` to find out how to catch these failures.)

Each active block is split into SIMD ("Single Instruction Multiple Data") groups of threads called "warps". Each warp contains the same number of threads, called the "warp size", which are executed by the multiprocessor in a SIMD fashion. This means each thread within a warp is broadcast the same instruction from the instruction store, which directs the thread to perform some operation or manipulation of local and/or global memory. The SIMD model is efficient and cost effective from a hardware standpoint, but from a software standpoint it unfortunately serializes conditional operations (e.g., both branches of the conditional must be evaluated). Be aware that conditional operations can have profound effects on the runtime of your kernels. With care this is generally a manageable problem but it can be problematic for some problems.

Active warps (that is, all the warps from all active blocks) are time-sliced: The thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources. The order of execution of the warps within a block and of blocks themselves is undefined, which means they can occur in any order. However, threads can be synchronized with `__syncthreads()`. Be aware that only after the execution of `__syncthreads()` are writes to shared (and global) memory guaranteed to be visible. Unless the variable is declared as volatile, the compiler is free to optimize (that is, reorder or eliminate) memory reads and writes to increase performance. The `__syncthreads()` call is allowed inside the scope of a conditional, but only if the conditional evaluates identically across the entire thread block. If not, the code execution is likely to hang or produce unintended side effects. Happily, `__syncthreads()` has low overhead as it only takes four (4) clock cycles to issue for a warp so long as no other thread has to wait for any other thread. A half-warp is either the first or second half of a warp, which is an important concept for memory accesses including coalescing memory accesses as discussed later in this article.

There are several take away messages from the previous discussion:

- Multiprocessor resources such as shared memory are limited and valuable.
- Effectively managing limited multiprocessor resources, such as shared memory, for a range of CUDA-enabled device configurations is a fact of life for CUDA developers.
- Be aware that conditional operations (such as `if` statements) can have a profound effect on the runtime of your kernels.
- The CUDA occupancy calculator and nvcc compiler are important tools to learn and use -- especially when exploring execution configurations.

The CUDA Memory Model

Figure 1 schematically illustrates a thread that executes on the device has access to global memory and the on-chip memory through the memory types.

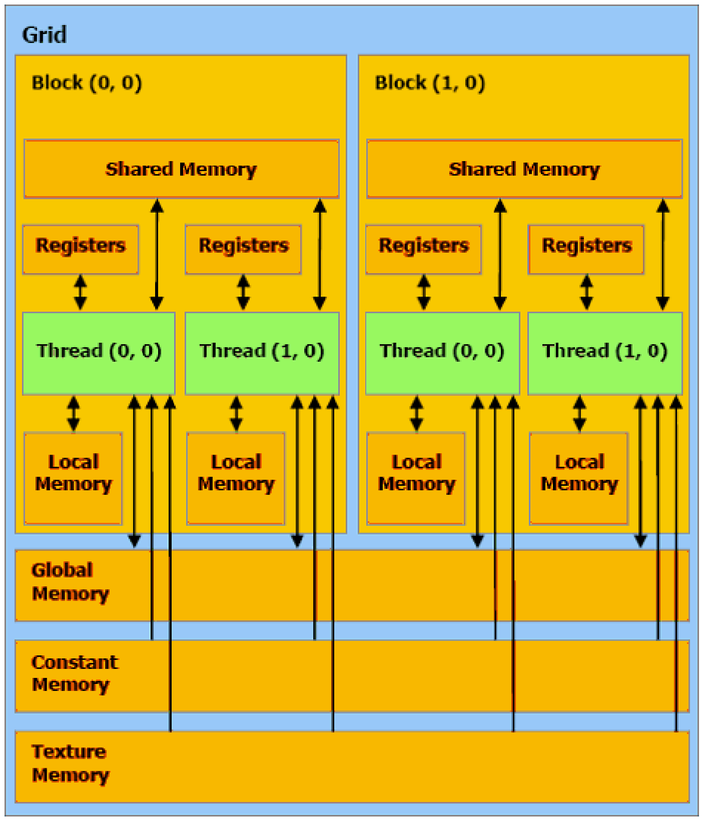

Figure 1

Each multiprocessor, illustrated as Block (0, 0) and Block (1, 0) above, contains the following four memory types:

- One set of local registers per thread.
- A parallel data cache or shared memory that is shared by all the threads and implements the shared memory space.
- A read-only constant cache that is shared by all the threads and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory. (Constant memory will be discussed in a later column. Until then, please refer to section 5.1.2.2 of the CUDA Programming Guide for more information.)
- A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory. (Texture memory will be discussed in a subsequent article. Until then, refer to section 5.1.2.3 of the CUDA Programming Guide for more information.)

Don't be confused by the fact the illustration includes a block labeled "local memory" within the multi-processor. Local memory implies "local in the scope of each thread". It is a memory abstraction, not an actual hardware component of the multi-processor. In actuality, local memory gets allocated in global memory by the compiler and delivers the same performance as any other global memory region. Local memory is basically used by the compiler to keep anything the programmer considers local to the thread but does not fit in faster memory for some reason. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. In some cases the compiler might choose to place these variables local memory, which might be the case when there are too many register variables, an array contains more than four elements, some structure or array would consume too much register space, or when the compiler cannot determine if an array is indexed with constant quantities.

Be careful because local memory can cause slow performance. Inspection of the ptx assembly code (obtained by compiling with the `-ptx` or `-keep` option) will tell if a variable has been placed in local memory during the first compilation phases as it will be declared using the `.local` mnemonic and accessed using the `ld.local` and `st.local` mnemonics. If it has not, subsequent compilation phases might still decide otherwise though if they find it consumes too much register space for the targeted architecture.

Until the next column installment, I recommend using the occupancy calculator to get a solid understanding of how the execution model and kernel launch execution configuration affects the number of registers and amount of shared memory.

For More Information

- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- 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

Click here for more information on CUDA and here for more information on NVIDIA.

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.