Using Shared Memory in CUDA C/C++

By Mark Harris, posted Jan 28 2013 at 11:21PM

RSS Feed

Tags: CUDA C, memory, Shared Memory

In the *previous post*, I looked at how global memory accesses by a group of threads can be coalesced into a single transaction, and how alignment and stride affect coalescing for various generations of CUDA hardware. For recent versions of CUDA hardware, misaligned data accesses are not a big issue. However, striding through global memory is problematic regardless of the generation of the CUDA hardware, and would seem to be unavoidable in many cases, such as when accessing elements in a multidimensional array along the second and higher dimensions. However, it is possible to coalesce memory access in such cases if we use shared memory. Before I show you how to avoid striding through global memory in the next post, first I need to describe shared memory in some detail.

Shared Memory

Because it is on-chip, shared memory is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency (provided that there are no bank conflicts between the threads, which we will examine later in this post). Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. This capability (combined with thread synchronization) has a number of uses, such as user-managed data caches, high-performance cooperative parallel algorithms (parallel reductions, for example), and to facilitate global memory coalescing in cases where it would otherwise not be possible.

Thread Synchronization

When sharing data between threads, we need to be careful to avoid race conditions, because while

threads in a block run *logically* in parallel, not all threads can execute *physically* at the same time. Let's say that two threads A and B each load a data element from global memory and store it to shared memory. Then, thread A wants to read B's element from shared memory, and vice versa. Let's assume that A and B are threads in two different warps. If B has not finished writing its element before A tries to read it, we have a race condition, which can lead to undefined behavior and incorrect results.

To ensure correct results when parallel threads cooperate, we must synchronize the threads. CUDA provides a simple barrier synchronization primitive, `__syncthreads()`. A thread's execution can only proceed past a `__syncthreads()` after all threads in its block have executed the `__syncthreads()`. Thus, we can avoid the race condition described above by calling `__syncthreads()` after the store to shared memory and before any threads load from shared memory. It's important to be aware that calling `__syncthreads()` in divergent code is undefined and can lead to deadlock—all threads within a thread block must call `__syncthreads()` at the same point.

Shared Memory Example

Declare shared memory in CUDA C/C++ device code using the `__shared__` variable declaration specifier. There are multiple ways to declare shared memory inside a kernel, depending on whether the amount of memory is known at compile time or at run time. The following complete code (*available on GitHub*) illustrates various methods of using shared memory.

```
#include <stdio.h>

__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }

  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)\n", i, i, d

  // run dynamic shared memory version
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
  cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)\n", i, i, d
}
```

This code reverses the data in a 64-element array using shared memory. The two kernels are very similar, differing only in how the shared memory arrays are declared and how the kernels are invoked.

Static Shared Memory

If the shared memory array size is known at compile time, as in the staticReverse kernel, then we can explicitly declare an array of that size, as we do with the array `s`.

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
```

```
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

In this kernel, `t` and `tr` are the two indices representing the original and reverse order, respectively. Threads copy the data from global memory to shared memory with the statement `s[t] = d[t]`, and the reversal is done two lines later with the statement `d[t] = s[tr]`. But before executing this final line in which each thread accesses data in shared memory that was written by another thread, remember that we need to make sure all threads have completed the loads to shared memory, by calling `__syncthreads()`.

The reason shared memory is used in this example is to facilitate global memory coalescing on older CUDA devices (Compute Capability 1.1 or earlier). Optimal global memory coalescing is achieved for both reads and writes because global memory is always accessed through the linear, aligned index `t`. The reversed index `tr` is only used to access shared memory, which does not have the sequential access restrictions of global memory for optimal performance. The only performance issue with shared memory is bank conflicts, which we will discuss later. (Note that on devices of Compute Capability 1.2 or later, the memory system can fully coalesce even the reversed index stores to global memory. But this technique is still useful for other access patterns, as I'll show in the next post.)

Dynamic Shared Memory

The other three kernels in this example use dynamically allocated shared memory, which can be used when the amount of shared memory is not known at compile time. In this case the shared memory allocation size per thread block must be specified (in bytes) using an optional third execution configuration parameter, as in the following excerpt.

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
```

The dynamic shared memory kernel, `dynamicReverse()`, declares the shared memory array using an unsized extern array syntax, `extern __shared__ int s[]` (note the empty brackets and use of the extern specifier). The size is implicitly determined from the third execution configuration parameter when the kernel is launched. The remainder of the kernel code is identical to the `staticReverse()` kernel.

What if you need multiple dynamically sized arrays in a single kernel? You must declare a single `extern` unsized array as before, and use pointers into it to divide it into multiple arrays, as in the following excerpt.

```
extern __shared__ int s[];
int *integerData = s;                // nI ints
float *floatData = &integerData[nI]; // nF floats
char *charData = &floatData[nF];     // nC chars
```

In the kernel launch, specify the total shared memory needed, as in the following.

```
myKernel<<<gridSize, blockSize, nI*sizeof(int)+nF*sizeof(float)+nC*siz
```

Shared memory bank conflicts

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. Therefore, any memory load or store of *n* addresses that spans *b* distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is *b* times as high as the bandwidth of a single bank.

However, if multiple threads' requested addresses map to the same memory bank, the accesses are serialized. The hardware splits a conflicting memory request into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of colliding memory requests. An exception is the case where all threads in a warp address the same shared memory address, resulting in a broadcast. Devices of compute capability 2.0 and higher have the additional ability to multicast shared memory accesses, meaning that multiple accesses to the same location by any number of threads within a warp are served simultaneously.

To minimize bank conflicts, it is important to understand how memory addresses map to memory banks. Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle. For devices of compute capability 1.x, the warp size is 32 threads and the number of banks is 16. A shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. Note that no bank conflict occurs if only one memory location per bank is accessed by a half warp of threads.

For devices of compute capability 2.0, the warp size is 32 threads and the number of banks is also 32. A shared memory request for a warp is not split as with devices of compute capability 1.x, meaning that bank conflicts can occur between threads in the first half of a warp and threads in the second half of the same warp.

Devices of compute capability 3.x have configurable bank size, which can be set using *cudaDeviceSetSharedMemConfig*() to either four bytes (cudaSharedMemBankSizeFourByte, the default) or eight bytes (`cudaSharedMemBankSizeEightByte`). Setting the bank size to eight bytes can help avoid shared memory bank conflicts when accessing double precision data.

Configuring the amount of shared memory

On devices of compute capability 2.x and 3.x, each multiprocessor has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory. For devices of compute capability 2.x, there are two settings, 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used. This can be configured during runtime API from the host for all

kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`. These accept one of three options: `cudaFuncCachePreferNone`, `cudaFuncCachePreferShared`, and `cudaFuncCachePreferL1`. The driver will honor the specified preference except when a kernel requires more shared memory per thread block than available in the specified configuration. Devices of compute capability 3.x allow a third setting of 32KB shared memory / 32KB L1 cache which can be obtained using the option `cudaFuncCachePreferEqual`.

Summary

Shared memory is a powerful feature for writing well optimized CUDA code. Access to shared memory is much faster than global memory access because it is located on chip. Because shared memory is shared by threads in a thread block, it provides a mechanism for threads to cooperate. One way to use shared memory that leverages such thread cooperation is to enable global memory coalescing, as demonstrated by the array reversal in this post. By reversing the array using shared memory we are able to have all global memory reads and writes performed with unit stride, achieving full coalescing on any CUDA GPU. In the next post I will continue our discussion of shared memory by using it to optimize a matrix transpose.

∥∀

Nsight Visual Studio Edition 3.0 New Features

PreviousPauseNext

Tag Index

accelerometer (1) Algorithms (3) Android (1) ANR (1) ARM (2) ArrayFire (1) Audi (1) Automotive & Embedded (1) Blog (21) Blog (23) Blog (1) Cluster (4) competition (1) Compilation (1) Concurrency (2) Copperhead (1) CUDA (26) CUDA 4.1 (1) CUDA 5.5 (3) CUDA C (15) CUDA Fortran (10) CUDA Pro Tip (1) CUDA Profiler (1) CUDA Spotlight (1) CUDA Zone (85) CUDACasts (2) Debug (1) Debugger (1) Debugging (4) depth of field (1) Develop 4 Shield (2) development kit (1) DirectCompute (1) DirectX (3) dynamic shadows (1) Eclipse (1) Events (2) FFT (1) Finite Difference (4) Floating Point (2) Game & Graphics Development (40) Games (1) Games and Graphics (11) GeForce Developer Stories (2) getting started (1) google io (1) GTC (2) Hardware (1) HDR (1) Interview (1) Kepler (1) Lamborghini (1) Libraries (4) memory (6) Mobile (1) Mobile Development (29) Monte Carlo (1) MPI (2) Multi-GPU (3) native_app_glue (1) NDK (1) NPP (1) Nsight (2) NSight Eclipse Edition (1) Nsight Tegra (2) NSIGHT Visual Studio Edition (2) Nsight Visual Studio Edition (1) NumbaPro (3) Numerics (1) NVIDIA Parallel Nsight (1) NVIDIA Shield (2) nvidia-smi (1) Occupancy (1) OpenACC (6) OpenGL (3) OpenGL ES (1) OTA (1) Parallel Forall (72) Parallel Nsight (1) Parallel Programming (5) PerfHUD ES (2) PerfHUDES (1) Performance (4) Portability (1) Porting (1) Pro Tip (5) Professional Graphics (6) Profiling (5) Programming Languages (1) Python (5) Robotics (1) Shader debugging (1) Shape Sensing (1) Shared Memory (6) Shield (2) Streams (2) tablet (1) TADP (2) Technologies (3) tegra (6) Tegra Android Developer Pack (1) Tegra Android Development Pack (1) Tegra Developer Stories (2) Tegra Profiler (2) Tegra Zone (1) Textures (1) Thrust (3) Tools (11) tools (3) Toradex (1) Visual Studio (3) Windows 8 (1) Windows 8.1 (1) xoom (1) Zone In (1)

Developer Blogs

Parallel Forall Blog
