# CUDA, Supercomputing for the Masses: Part 5

Understanding and using shared memory (2)

June 30, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/208801731

In Part 4 of this article series on CUDA, I discussed how the execution model and kernel launch execution configuration affects the number of registers and amount of local multiprocessor resources such as shared memory. In this installment, I continue with a discussion of memory performance and the use of shared memory in reverseArray_multiblock_fast.cu.

## CUDA Memory Performance

The local and global memory spaces are not cached which means each memory access to global memory (or local memory) generates an explicit memory access. So what does it cost to access (read or write, for example) each of the different memory types?

A multiprocessor takes four clock cycles to issue one memory instruction for a "warp". Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency. As an example, the assignment operator in the code snippet below takes four clock cycles to issue a `read` from global memory, four clock cycles to issue a `write` to shared memory, and 400 to 600 clock cycles to read a `float` from global memory. Note: the `__device__` variable type qualifier is used to denote a variable that resides in global memory (among other variable characteristics; see section 4.2.2.1 of the CUDA Programming Guide for more information). Variables of type `__device__` cannot be accessed by host code.

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

With a factor of 100x-150x difference in access time, it is no surprise that developers need to minimize accesses to global memory and reuse data within the local multiprocessor memories. The CUDA designers have done a good job with the thread scheduler; so much of the global memory latency can be transparently hidden just by specifying large numbers of blocks in the execution configuration and working as much as possible with variables with register, `__shared__`, and `__constant__` memory types in the kernel.

Since shared memory is on chip accesses are significantly faster than accesses to global memory and the main optimization is avoiding bank conflicts. Shared memory is fast (some documentation indicates it is as fast as register accesses). However, recent large improvements in CUBLAS and CUFFT performance were achieved by avoiding shared memory in favor of registers -- so try to use registers whenever possible. CUDA shared memory is divided into equally-sized memory modules that are called memory banks. Each memory bank holds a successive 32-bit value (like an `int` or `float`) so consecutive array accesses by consecutive threads are very fast. Bank conflicts occur when multiple requests are made for data from the same bank (either the same address or multiple addresses that map to the same bank). When this happens, the hardware effectively serializes the memory operations, which forces all the threads to wait until all the memory requests are satisfied. If all threads read from the same shared memory address then a broadcast mechanism is automatically invoked and serialization is avoided. Shared memory broadcasts are an excellent and high-performance way to get data to many threads simultaneously. It is worthwhile trying to exploit this feature whenever you use shared memory.

I will discuss bank conflicts in greater detail in a future column. For the moment, suffice it to say that reverseArray_multiblock_fast.cu has no bank conflicts because consecutive threads access consecutive values.

A quick summary of local multiprocessor memory types with read/write capability follows:

- Registers:
    - The fastest form of memory on the multi-processor.
    - Is only accessible by the thread.
    - Has the lifetime of the thread.
- Shared Memory:
    - Can be as fast as a register when there are no bank conflicts or when reading from the same address.
    - Accessible by any thread of the block from which it was created.
    - Has the lifetime of the block.
- Global memory:
    - Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes which will be discussed in the next column.
    - Accessible from either the host or device.
    - Has the lifetime of the application.
- Local memory:

  - A potential performance gotcha, it resides in global memory and can be 150x slower than register or shared memory.
  - Is only accessible by the thread.
  - Has the lifetime of the thread.

## Shared Memory Cautions

- Watch out for shared memory bank conflicts, which can slow performance.
- All dynamically allocated shared variables in a kernel start at the same memory address. Using more than one dynamically allocated shared memory array requires manually generating the offset. For example, if you want dynamically allocated shared memory to contain two arrays, `a` and `b`, then you need to do something like:

```
__global__ void kernel(int aSize)
{
   extern __shared__ float sData[];
   float *a, *b;

   a = sData;
   b = &a[aSize];
```

## Register/Local Memory Cautions

- Register memory can be transparently placed into local memory. This can potentially be a cause for poor performance. Check the `ptx` assembly code or look for `lmem` in the output from `nvcc` with the `"-ptxas-options=-v"`.
- Arrays indexed by constants known at compile time typically reside in registers but if they are indexed by variables they cannot reside in registers. This creates a conundrum for the developer because loop unrolling may be required to keep array elements in register memory as opposed to slow global memory. However, unrolling loops can greatly increase register usage, which may result in variables being kept in local memory -- obviating any benefit of loop unrolling. It is possible to use the nvcc option, `-maxrregcount=value` to tell the compiler to use more registers. (Note: the maximum register count that can be specified is 128.) This is a tradeoff between using more registers and creating fewer threads, which may hinder the opportunities to hide memory latency. With some architectures, use of this option may also prevent kernels from starting due to insufficient resources.

## A Shared Memory Kernel

Both programs reverseArray_multiblock.cu and revereseArray_multiblock_fast.cu perform the same tasks. They create a 1D array of integers, `h_a`, containing the integer values `[0 .. dimA-1]`. The array is then moved via `cudaMemcpy` to the device and the host then launches the `reverseArrayBlock` kernel to reverse order the array contents in place. Again, `cudaMemcpy` is used to transfer data from the device to the host where a check is performed to verify that the device produced the correct result (for example, `[dimA-1 .. 0]`).

The difference is that reverseArray_multiblock_fast.cu uses shared memory to improve the performance of the kernel, while reverseArray_multiblock.cu operates entirely in global memory. Try timing the two programs and verify for yourself the difference in performance. Also, reverseArray_multiblock.cu accesses global memory in an inefficient manner. We will use the CUDA profiler to help diagnose and fix this performance issue in a future column, and show how improvements in the new 10 series architecture eliminate the need for these types of optimizations in many cases.

```
// includes, system
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

// Part 2 of 2: implement the fast kernel using shared memory
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    extern __shared__ int s_data[];

    int inOffset  = blockDim.x * blockIdx.x;
    int in  = inOffset + threadIdx.x;

    // Load one element per thread from device memory and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];

    // Block until all threads in the block have written their data to shared mem
    __syncthreads();

    // write the data from shared memory in forward order,
    // but to the reversed block offset as before

    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);

    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
}

////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
```

```
{
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)

    // pointer for device memory
    int *d_b, *d_a;

    // define grid and block size
    int numThreadsPerBlock = 256;

    // Compute number of blocks needed based on array size and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // Part 1 of 2: Compute the number of bytes of shared memory needed
    // This is used in the kernel invocation below
    int sharedMemSize = numThreadsPerBlock * sizeof(int);

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc( (void **) &d_a, memSize );
    cudaMalloc( (void **) &d_b, memSize );

    // Initialize input array on host
    for (int i = 0; i < dimA; ++i)
    {
        h_a[i] = i;
    }

    // Copy host array to device array
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

    // launch kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid, dimBlock, sharedMemSize >>>( d_b, d_a );

    // block until the device has completed
    cudaThreadSynchronize();

    // check if kernel execution generated an error
    // Check for any CUDA errors
    checkCUDAError("kernel invocation");

    // device to host copy
    cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

    // Check for any CUDA errors
    checkCUDAError("memcpy");

    // verify the data returned to the host is correct
    for (int i = 0; i < dimA; i++)
    {
        assert(h_a[i] == dimA - 1 - i );
    }

    // free device memory
    cudaFree(d_a);
    cudaFree(d_b);

    // free host memory
    free(h_a);

    // If the program makes it this far, then the results are correct and
    // there are no run-time errors.  Good work!
    printf("Correct!\n");

    return 0;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}
```

Deciding on the amount of shared memory at runtime requires some setup in both host and device code. In this example , the amount of shared memory (in bytes) for each block in a kernel is specified in the execution configuration on the host as an optional third parameter. (Setup on the host side is only required if the amount of shared memory is specified at kernel launch. If it's fixed at compile time no setup is required on the host side.) By default, the execution configuration assumes no shared memory is used. For example, in the host code of arrayReversal_multiblock_fast.cu, the following code snippet allocates shared memory for an array of integers containing a number of elements equal to the number of threads in a block:

```
// Part 1 of 2: Compute the number of bytes of share memory needed
// This is used in the kernel invocation below
int sharedMemSize = numThreadsPerBlock * sizeof(int);
```

Looking at the `reverseArrayBlock` kernel, the shared memory is declared with the following:

```
extern __shared__ int s_data[];
```

Note that the size is not indicated in the kernel -- rather it is obtained from the host through the execution configuration.

Until the next column on profiling, I recommend looking at the reverseArray_multiblock.cu. Do you think there is a performance problem in accessing global memory? If you think there is a problem, try to fix it.

## 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, here for the the CUDA Occupancy Calculator, 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.*