CUDA, Supercomputing for the Masses: Part 3

Error handling and global memory performance limitations

May 13, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207603131

Congratulations! Thanks to Part 1 and Part 2 of this series on CUDA (short for "Compute Unified Device Architecture"), you are now a CUDA-enabled programmer with the ability to create and run programs that can use many hundreds of simultaneous threads on CUDA-enabled devices. In Part 2, incrementArrays.cu, I provided a working example of a pervasive CUDA application pattern -- move data to device, run one or mores kernels to perform a calculation, and retrieve the result(s). Essentially, incrementArrays.cu can be morphed into whatever application you desire simply by substituting your own kernel and loading your own data (which is what I do for the example in this column). Subsequent columns will discuss CUDA asynchronous I/O and streams.

"You now know enough to be dangerous!" is a humorous and accurate way to summarize the previous paragraph. The good news about CUDA is that it provides a natural way to translate your thoughts as a programmer into massively-parallel programs. The bad news is that more understanding is required to make those programs both robust and efficient.

Don't be cautious. Start experimenting and go for it! CUDA provides both programming tools and constructs to create excellent software and the only way to really learn it is to experiment. In reality, these columns will supplement your experimentation and learning process by highlighting CUDA features through short examples and by bringing good sources of information on the Internet to your attention. Remember that the CUDA Zone is a great place for all things CUDA and the forums are a great place to look for answers to your questions, plus they have the advantage of being interactive so you can post questions and get answers.

This and the next few columns utilize a simple array reversal application to expand your knowledge and highlight the performance impact of shared memory. I discuss error checks and performance behavior, along with the CUDA profiling tool. I've also included the source listing for the next column so you can see how to implement array reversal with shared memory. The program reverseArray_multiblock.cu implements an obvious, yet low performance, way to reverse an array in global memory on a CUDA device. Do not use it as a model for your applications because global memory is not the best memory type to use for this application -- plus this version also performs uncoalesced memory accesses, which adversely affects global memory performance. The best global memory bandwidth is achieved when simultaneous memory accesses can be coalesced into a single memory transaction. In subsequent columns, I discuss the differences between global and shared memory as well as the various requirements for memory accesses to coalesce based on the compute capability of the device.

CUDA Error Handling

Detecting and handling errors is essential to creating robust and usable software. Users tend to get very grumpy when their applications fail or produce incorrect results. For developers, adding error-handling code can be annoying, and tedious. It can clutter up the elegance of the code, and slow the development process in attempting to deal with every conceivable error. Yes, error handling is a thankless job but keep in mind you are not doing it for yourself (although I have been saved countless times through good error checking) -- rather it is being done for the people who are going to use the program. If something can fail, users need to know why it failed and, more importantly, what they can do to fix the problem. Enough said, good error-handling and recovery can really make your application a hit with the users. Commercial developers should especially take note.

The CUDA designers are aware of the importance of good error handling. To facilitate this, every CUDA call -- with the exception of kernel launches -- returns an error code of type `cudaError_t`. Upon successful completion, `cudaSuccess` is returned. Otherwise, an error code is returned.

A human-readable description of the error can be obtained from:

```
char *cudaGetErrorString(cudaError_t code);
```

C-language programmers will recognize a similarity between this method and the C library, which uses the variable `errno` to indicate errors and the reporting of human-readable error messages with `perror` and `strerror`. The C library paradigm has worked well for many millions of lines of C-code, and there is no doubt it should work well in the future for CUDA software.

CUDA also provides a method, `cudaGetLastError`, which reports the last error for any previous runtime call in the host thread. This has multiple implications:

- The asynchronous nature of the kernel launches precludes explicitly checking for errors with `cudaGetLastError`.
  Instead, use `cudaThreadSynchronize` which blocks until the device has completed all previous calls, including kernel calls, and returns an error if one of the preceding tasks fails. Queuing multiple kernel launches unfortunately implies that error checking can only be done after all the kernels have completed -- unless explicit error checking and reporting to the host is performed by programmers within the kernel.
- Errors are reported to the correct host thread. If the host is running multiple threads, as might be the case when an application is using multiple CUDA

devices, the error will be reported to the correct host thread.
- When multiple errors occur between calls to `cudaGetLastError`, only the last error will be reported. This means the programmer must take care to tie the error to the runtime call that generated the error or risk making an incorrect error report to users.

Looking at the Source Code

Looking at the source code for reverseArray_multiblock.cu, you notice that the structure of the program is very, very similar to the structure of moveArrays.cu from Part 2. An error routine, `checkCUDAError` is provided so the host can print out a human-readable message and exit when an error is reported by `cudaGetLastError`. As can be seen, `checkCUDAError` is judiciously utilized throughout the program to check for errors.

The program reverseArray_multiblock.cu essentially creates a 1D array of integers, `h_a`, containing the integer values `[0 .. dimA-1]`. Array `h_a` is moved via `cudaMemcpy` to array `d_a`, which resides in global memory on the device. The host then launches the `reverseArrayBlock` kernel to copy the array contents in reverse order from `d_a` to `d_b`, which is another global memory array. Again, `cudaMemcpy` is used to transfer data -- this time from `d_b` to the host. A check is then performed on the host to verify that the device produced the correct result (e.g, `[dimA-1 .. 0]`).

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

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

// Part3: implement the kernel
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset  = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in  = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}
////////////////////////////////////////////////////////////////////
// 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;

    // Part 1: compute number of blocks needed based on
    // array size and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // 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 >>>( 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);
    }
}
```

A key design feature of this program is that both arrays `d_a` and `d_b` reside in global memory on the device. The CUDA SDK provides an example program, bandwidthTest, which provides some information about the device characteristics. On my system, the global memory bandwidth is slightly over 60 GB/s. This is excellent until you consider that this bandwidth must service 128 hardware threads -- each of which can deliver a large number of floating-point operations. Since a 32-bit floating-point value occupies four (4) bytes, global memory bandwidth limited applications on this hardware will only be able to deliver around 15 GF/s -- or only a small percentage of the available performance capability. (This assumes the application only reads from global memory and does not write to it.) Obviously, higher performance applications must reuse data in some fashion. This is the function of shared and register memory and it is our job as programmers to gain the maximum benefit of these memory types. To gain a better understanding of machine balance as floating-point capability relates to memory bandwidth (and other machine characteristics), read my article HPC Balance and Common Sense.

Shared Memory Version

The following source listing is for arrayReversal_multiblock_fast.cu, which I discuss in the next installment. I provide it now for your convenience so you can see how to use shared memory on this problem right now.

```
// 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 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 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);
    }
}
```

In the next column, I begin looking at the use of shared memory to increase performance. Until then, I suggest looking into the CUDA memory types -- specifically `__shared__`, `__constant__`, and `register 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.