Architecture & Design

July 25, 2008

CUDA, Supercomputing for the Masses: Part 6

(Page 1 of 2)

- Email
- Print
- Reprint

add to:
- Del.icio.us
- Slashdot
- Digg
- Y! MyWeb
- Google
- Blink
- Furl

TABLE OF CONTENTS

Rob Farber

Global memory and the CUDA profiler

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.

In Part 5 of this article series on CUDA (short for "Compute Unified Device Architecture"), I discussed memory performance and the use of shared memory in reverseArray_multiblock_fast.cu. In this installment, I examine global memory using the CUDA profiler

Astute readers of this series timed the two versions of the reverse array example discussed in Part 4 and Part 5 and were puzzled about how the shared memory version is faster than the global memory version. Recall that the shared memory version, reverseArray_multiblock_fast.cu, kernel copies array data from the global memory to the shared memory, then back to global memory while the slower kernel, reverseArray_multiblock.cu, only copies data from global memory to global memory. Since global memory performance is between 100x-150x slower than shared memory, shouldn't the significantly slower global memory performance dominate the runtime of both examples? Why is the shared memory version faster?

Answering this question requires understanding more about global memory plus the use of additional tools from the CUDA development environment -- specifically the CUDA profiler. Profiling CUDA software is fast and easy, as both the text and visual versions of the profiler read hardware profile counters on CUDA-enabled devices. Enabling text profiling is as easy as setting the environmental variables that start and control the profiler. Using the visual profiler is equally easy: Just start cudaprof and start clicking in the GUI. Profiling provides valuable insight. The collection of profile events is handled entirely by hardware within CUDA enabled devices. However, profiled kernels are no longer asynchronous. Reporting of results to the host only occurs after each kernel completes, which minimizes any communications impact.

Global Memory

Understanding how to efficiently use global memory is an essential requirement to becoming an adept CUDA programmer. Following is a brief discussion about global memory that should be sufficient to understand the performance difference between reverseArray_multiblock.cu and reverseArray_multiblock_fast.cu. Future columns will, of necessity, continue to explore efficient uses of global memory. In the meantime, a detailed discussion on global memory, with illustrations, can be found in Section 5.1.2.1 of the CUDA Programming Guide.

Global memory delivers the highest memory bandwidth only when the global memory accesses can be coalesced within a half-warp so the hardware can then fetch (or store) the data in the fewest number of transactions. CUDA Compute Capability devices (1.0 and 1.1) can fetch data in a single 64-byte or 128-byte transaction. If the memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp, which is undesirable. The performance penalty for non-coalesced memory operations varies according to the size of the data type. The CUDA documentation provides some rough guidelines for the expected performance degradation to expect for various size data types:

- 32-bit data types will be roughly 10x slower
- 64-bit data types will be roughly 4x slower
- 128-bit data types will be roughly 2x slower

Global memory access by all threads in the half-warp of a block can be coalesced into efficient memory transactions on a G80 architecture when:

1. The threads access 32-, 64- or 128-bit data types.
2. All 16 words of the transaction must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words). This implies that the starting address and alignment are important.
3. Threads must access the words in sequence: the kth thread in the half-warp must access the *k*th word. Note: not all threads in a warp need to access memory for the thread accesses to coalesce. This is called a "divergent warp".

Newer architectures such as the GT200 family of devices have more relaxed coalescing requirements than those just discussed. I will discuss architectural differences more deeply in a future column. For purposes here, suffice to say that if you tune your code to coalesce well on a G80 CUDA-enabled device, it will coalesce well on a GT200 device.

1 Introduction | 2 Enabling and Controlling Profiling Next Page

Architecture & Design

July 25, 2008

Email Print
Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Furl

CUDA, Supercomputing for the Masses: Part 6

(Page 2 of 2)

TABLE OF CONTENTS

Enabling and Controlling Textual Profiling

The environmental variables that control the text version of the CUDA profiler are:

- CUDA_PROFILE: Set to 1 or 0 to enable/disable the profiler
- CUDA_PROFILE_LOG: Set to the name of the log file (The default is ./cuda_profile.log)
- CUDA_PROFILE_CSV: Set to 1 or 0 to enable or disable a comma separated version of the log
- CUDA_PROFILE_CONFIG: Specify a configuration file with up to four signals

The last bullet is important because only four signals can be profiled at a time. The developer can have the profiler collect any of the following events by specifying their names on separate lines in the file named by CUDA_PROFILE_CONFIG:

- **gld_incoherent**: Number of non-coalesced global memory loads
- **gld_coherent**: Number of coalesced global memory loads
- **gst_incoherent**: Number of non-coalesced global memory stores
- **gst_coherent**: Number of coalesced global memory stores
- **local_load**: Number of local memory loads
- **local_store**: Number of local memory stores
- **branch**: Number of branch events taken by threads
- **divergent_branch**: Number of divergent branches within a warp
- **instructions**: instruction count
- **warp_serialize**: Number of threads in a warp that serialize based on address conflicts to shared or constant memory
- **cta_launched**: executed thread blocks

Notes on Profiler Counters

Note that the performance counter values do not correspond to individual thread activity. Instead, these values represent events within a thread warp. For example, an incoherent store within a thread warp will increment the **gst_incoherent** counter by 1. So the final counter value stores information for all incoherent stores in all warps.

In addition, the profiler can only target one of the multiprocessors in the GPU, so the counter values will not correspond to the total number of warps launched for a particular kernel. For this reason, when using the performance counter options in the profiler the user should always launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work. In practice, NVIDIA suggests it is best to launch at least 100 blocks or so for consistent results.

As a result, users should not expect the counter values to match the numbers one would determine through inspection of the kernel code. Counter values are best used to identify relative performance differences between unoptimized and optimized code. For example, if the profiler reports some number of non-coalesced global loads for an initial piece of software, then it is easy to see if a more refined version of the code utilizes a smaller number of non-coalesced loads. In most cases, the goal is to make the number of non-coalesced global loads zero, so the counter value is useful for tracking progress toward this goal.

Profiling Results

Let's look at reverseArray_multiblock.cu and reverseArray_multiblock_fast.cu with the profiler. In this example, we will set the environment variables and configuration file in the bash shell under Linux as follows:

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=$HOME/.cuda_profile_config
```

Profiler configuration via environnent variables in Linux with bash

```
gld_coherent
gld_incoherent
gst_coherent
gst_incoherent
```

Contents of the CUDA_PROFILE_CONFIG file

Running the reverseArray_multiblock.cu executable generates the following profiler report in ./cuda_profile.log:

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,gst_coherent
method=[ memcopy ] gputime=[ 438.432 ]
method=[ _Z17reverseArrayBlockPiS_ ] gputime=[ 267.520 ] cputime=[ 297.000 ] occupancy=[ 1.000 ] gld_incoherent=[ 0 ] gld_coherent=
[ 1952 ] gst_incoherent=[ 62464 ] gst_coherent=[ 0 ]
method=[ memcopy ] gputime=[ 349.344 ]
```

Profile report for reverseArray_multiblock.cu

Similarly, running the reverseArray_multiblock_fast.cu executable produces the following output, which overwrites the previous output in .cuda_profile.log.

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,gst_coherent
method=[ memcopy ] gputime=[ 449.600 ]
method=[ _Z17reverseArrayBlockPiS_ ] gputime=[ 50.464 ] cputime=[ 108.000 ] occupancy=[ 1.000 ] gld_incoherent=[ 0 ] gld_coherent=
[ 2032 ] gst_incoherent=[ 0 ] gst_coherent=[ 8128 ]
method=[ memcopy ] gputime=[ 509.984 ]
```

Profile report for reverseArray_multiblock_fast.cu

Comparing these two profiler results shows that reverseArray_multiblock_fast.cu has zero incoherent stores as opposed to reverseArray_multiblock.cu, which has many. Look at the source of reverseArray_multiblock.cu and see if you can fix the performance problem with incoherent stores. Once fixed, measure how fast the two programs are relative to each other.

For convenience, Listing One presents reverseArray_multiblock.cu and Listing Two reverseArray_multiblock_fast.cu.

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

reverseArray_multiblock.cu

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

reverseArray_multiblock_fast.cu

For More Information

- 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.

Previous Page | 1 Introduction | 2 Enabling and Controlling Profiling