L3: Data Partitioning and Memory Organization, Synchronization

January 21, 2009

Outline

- More on CUDA
- First assignment, due Jan. 30 (later in lecture)
- Error checking mechanisms
- Synchronization
- More on data partitioning
- **Reading:** GPU Gems 2, Ch. 31; CUDA 2.0 Manual, particularly Chapters 4 and 5

This lecture includes slides provided by:
Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)
see http://courses.ece.uiuc.edu/ece498/al1/

and Austin Robison (NVIDIA)

Today's Lecture

- Establish background to write your first CUDA code
- Testing for correctness (a little)
- Debugging (a little)
- Core concepts to be reinforced
  - Data partitioning
  - Synchronization

Questions from Last Time

- Considering the GPU gets a single copy of the array, can two threads access the shared array at the same time?
  - Yes, if no bank conflicts. But watch out!
- Can we have different thread functions for different subsets of data?
  - Not efficiently.
- Why do I get the same results every time using a random number generator?
  - Deterministic if starts with the default (or same) seed.

My Approach to Working with New Systems

- Approach with caution
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that documentation is not always clear or accurate.
- Crawl, then walk, then run
  - Start with something known to work.
  - Only change one "variable" at a time.

Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

Run-time functions & macros for error checking

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or -D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))

check for error in run-time call and exit if error detected

CUT_SAFE_MALLOC(cudaMalloc(<args>));

similar to above, but for malloc calls

CUT_CHECK_ERROR("error message goes here");

check for error immediately following kernel execution and if detected, exit with error message

Simple working code example from last time

- Goal for this example:
  - Really simple but illustrative of key concepts
  - Fits in one file with simple compile command
  - Can absorb during lecture
- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does "6" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Known as a cyclic data distribution

Reductions (from last time)

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs? (next class and assignment)

Reminder: Gathering and Reporting Results

- Global, device functions and excerpts from host, main

```
__device__ int compare(int a, int b) {
  if (a == b) return 1;
  return 0;
}

__global__ void final_sum(int *d_in, int
  *d_out) {
  d_out[thre
  for (i=0; i<
    int val = d_in[i*BLOCKSIZE +
          threadIdx.x];
    d_out[threadIdx.x] +=
          compare(val, 6);
  }
}
```

Compute individual results for each thread

Serialize final resutls gathering on host

```
int __host__ void outer_compute
  (int *h_in_array, int *h_out_array) {
  …
  compute<<<1,BLOCKSIZE,msize)>>>
    (d_in_array, d_out_array);

  cudaMemcpy(h_out_array,
  d_out_array,
    BLOCKSIZE*sizeof(int),
    cudaMemcpyDeviceToHost);

main(int argc, char **argv) {
…
  for (int i=0; i<BLOCKSIZE; i++)
  { sum+=out_array[i]; }
  printf ("Result = %d\n",sum);
}
```


Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- Why is this needed?
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.
  - Allows the program to guarantee partial ordering of these accesses to prevent incorrect orderings.
- Watch out!
  - Potential for deadlock when it appears in conditionals

Gathering Results on GPU for "Count 6"

```
__global__ void compute(int *d_in, int
  *d_out) {
 d_out[threadIdx.x] = 0;
 for (i=0; i<SIZE/BLOCKSIZE; i++) {
    int val = d_in[i*BLOCKSIZE +
          threadIdx.x];
    d_out[threadIdx.x] +=
                compare(val, 6);
 }
}
```

```
__global__ void compute(int *d_in, int
  *d_out, int *d_sum) {
 d_out[threadIdx.x] = 0;
 for (i=0; i<SIZE/BLOCKSIZE; i++) {
    int val = d_in[i*BLOCKSIZE +
          threadIdx.x];
    d_out[threadIdx.x] +=
                compare(val, 6);
 }
 __synthreads();
 if (threadIdx.x == 0) {
  for 0..BKSIZE-1
  *d_sum += d_out[i];
 }
}
```


Also Atomic Update to Sum Variable

int atomicAdd(int* address, int val);

Increments the integer at address by val.

Atomic means that once initiated, the operation executes to completion without interruption by other threads

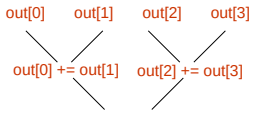
Example:

atomicAdd(d_sum, d_out_array[threadIdx.x]);

Programming Assignment #1

Problem 1:

In the "count 6" example using synchronization, we accumulated all results into out_array[0], thus serializing the accumulation computation on the GPU. Suppose we want to exploit some parallelism in this part, which will likely be particularly important to performance as we scale the number of threads. A common idiom for reduction computations is to use a tree-structured results-gathering phase, where independent threads collect their results in parallel. The tree below illustrates this for our example, with SIZE=16 and BLOCKSIZE=4.



Your job is to write this version of the reduction in CUDA. You can start with the sample code, but adjust the problem size to be larger:

```
#define SIZE 256
#define BLOCKSIZE 32
```


Another Example: Adding Two Matrices

CPU C program

```
void add_matrix_cpu(float *a, float *b,
  float *c, int N)
{
int i, j, index;
for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
      index =i+j*N;
      c[index]=a[index]+b[index];
    }
  }
}

void main() {
   .....
  add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu(float *a,
float *b, float *c, int N)
{
   int i =blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   int index =i+j*N;
   if( i <N && j <N)
     c[index]=a[index]+b[index];
}

void main() {
   dim3 dimBlock(blocksize,blocksize);
   dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);

add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c
,N);
}
```

Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

- Let blocksize = 2 and N = 4

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Closer Inspection of Computation and Data Partitioning

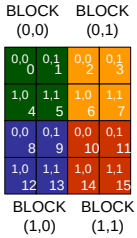
- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

- Let blocksize = 2 and N = 4



Data Distribution to Threads

- Many data parallel programming languages have mechanisms for expressing how a distributed data structure is mapped to threads
  - That is, the portion of the data a thread accesses (and usually stores locally)
  - Examples: HPF, Chapel, UPC
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to "data reuse" and communication needs
  - Affinity with other data
  - Locality: transfer size and capacity limit on shared memory
  - Access patterns relative to computation (avoid bank conflicts)
  - Minimizing overhead: Cost of copying to host and between global and shared memory

Concept of Data Distributions in CUDA

- This concept is not completely natural for CUDA
  - Implicit from computation partitioning and access expressions within threads
  - Within a block of threads, most memory is shared, so not really distributed
- Nevertheless, I think it is still useful
  - Spatial understanding of computation and data organization (helps conceptualize)
  - Distribution of data needed for objects too large to fit in shared memory (i.e., partitioning across blocks in a grid)
  - Helpful in putting CUDA in context of other languages

Common Data Distributions

- Consider a 1-Dimensional array

CYCLIC:

```
for (i = 0; i<BLOCKSIZE; i++)
        … d_in [i*BLOCKSIZE + threadIdx.x];
```

BLOCK:

```
for (i=threadIdx.x*BLOCKSIZE; i<(threadIdx.x+1)
     *BLOCKSIZE; i++)
   … d_in[i];
```

BLOCK-CYCLIC:

Programming Assignment #1, cont.

Problem 2: Computation Partitioning & Data Distribution

Using the matrix addition example from Lecture 2 class notes, develop a complete CUDA program for integer matrix addition. Initialize the two input matrices using random integers from 0 to 9 (as in the "count 6" example). Initialize the ith entry in both a[i] and b[i], so that everyone's matrix is initialized the same way.

Your CUDA code should use a data distribution for the input and output matrices that exploits several features of the architecture to improve performance (although the advantages of these will be greater on objects allocated to shared memory and with reuse of data within a block).

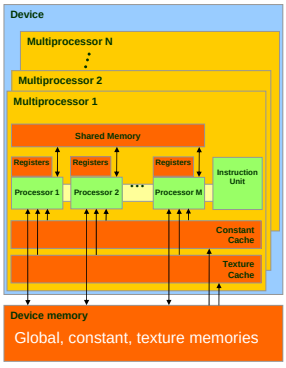
- a grid of 16 square blocks of threads (>= # multiproc)
- each block executing 64 threads (> warpsize)
- each thread accessing 32 elements (better if larger)

Within a grid block, you should use a BLOCK data distribution for the columns, so that each thread operates on a column. This should help reduce memory bank conflicts.

Return the output array.

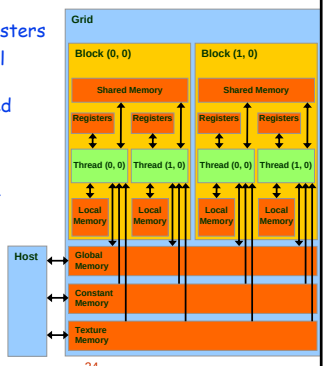
Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space



Programming Model: Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can read/write global, constant, and texture memory

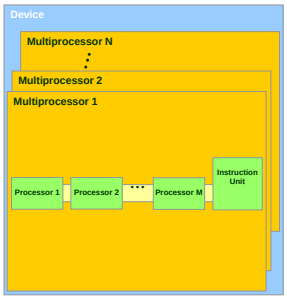


Threads, Warps, Blocks

- There are (up to) 32 threads in a Warp
  - Only <32 when there are fewer than 32 **total** threads
- There are (up to) 16 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- G80 has 16 SMs
- At least 16 Blocks required to "fill" the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

Hardware Implementation: A Set of SIMD Multiprocessors

- A device has a set of multiprocessors
- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture
  - Shared instruction unit
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
- The number of threads in a warp is the warp size



Hardware Implementation: Execution Model

- Each thread block of a grid is split into warps, each gets executed by one multiprocessor (SM)
  - The device processes only one grid at a time
- Each thread block is executed by one multiprocessor
  - So that the shared memory space resides in the on-chip shared memory
- A multiprocessor can execute multiple blocks concurrently
  - Shared memory and registers are partitioned among the threads of all concurrent blocks
  - So, decreasing shared memory usage (per block) and register usage (per thread) increases number of blocks that can run concurrently

Terminology Review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    `__global__ void KernelFunc(float* ptr)`
  - Obtained as the address of a global variable: `float* ptr = &GlobalVar;`

Relating this Back to the "Count 6" Example

- A few variables (i, val) are going in device registers
- We are copying input array into device and output array back to host
  - These device copies of the data all reside in global memory
- Suppose we want to access only shared memory on the devices
  - Cannot copy into shared memory from host!
  - Requires another copy from/to global memory to/from shared memory
- Other options
  - Use constant memory for input array (not helpful for current implementation, but possibly useful)
  - Use shared memory for output array and just return single result (today's version with synchronization)

Summary of Lecture

- Some error checking techniques to determine if program is correct
- Synchronization constructs
- Discussion of Data Partitioning
- First assignment, due FRIDAY, JANUARY 30, 5PM

Next Week

- Reasoning about parallelization
  - When is it safe?