L19: Advanced CUDA Issues

November 10, 2009

Administrative

- CLASS CANCELLED, TUESDAY, NOVEMBER 17
- Guest Lecture, November 19, Ganesh Gopalakrishnan
- Thursday, November 12
  - Turn in 1 page project description (next few slides)
  - handin cs4961 pdesc <file, ascii or PDF ok>
- VTUNE problem identified, solution coming soon
  - Has to do with where "Application Data" is redirected in accounts
- Mailing list: cs4961@list.eng.utah.edu

A Few Words About Final Project

- Purpose:
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Present results to work on communication of technical ideas
- Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations.
  - OpenMP + SSE-3
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - TBB + SSE-3
  - MPI + OpenMP
  - MPI + SSE-3
  - MPI + CUDA
- Present results in a poster session on the last day of class

Example Projects

- Look in the textbook or on-line
  - Recall Red/Blue from Ch. 4
    - Implement in MPI (+ SSE-3)
    - Implement main computation in CUDA
  - Algorithms from Ch. 5
  - SOR from Ch. 7
    - CUDA implementation?
  - FFT from Ch. 10
  - Jacobi from Ch. 10
  - Graph algorithms
  - Image and signal processing algorithms
  - Other domains…

Next Thursday, November 12

- Use handin program on CADE machines
  - handin cs4961 pdesc <file, ascii or PDF ok>
- Projects can be individual or group efforts, with 1 to three students per project.
- Turn in <1 page project proposal
  - Algorithm to be implemented
  - Programming model(s)
  - Implementation plan
  - Validation and measurement plan

Outline

- Reminder of CUDA Architecture
- Execution Model
  - Brief mention of control flow
- Heterogeneous Memory Hierarchy
  - Locality through data placement
  - Maximizing bandwidth through global memory coalescing
  - Avoiding memory bank conflicts
- Tiling and its Applicability to CUDA Code Generation

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)

Reading

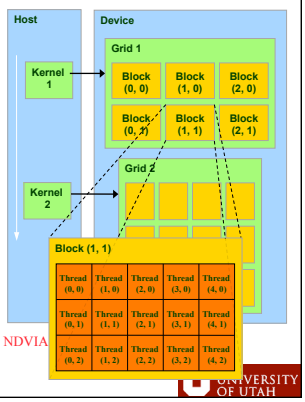
- David Kirk and Wen-mei Hwu manuscript (in progress)
  - http://www.toodoc.com/CUDA-textbook-by-David-Kirk-from-NVIDIA-and-Prof-Wen-mei-Hwu-pdf.html
- CUDA 2.x Manual, particularly Chapters 2 and 4 (download from nvidia.com/cudazone)
- Nice series from Dr. Dobbs Journal by Rob Farber
  - http://www.ddj.com/cpp/207200659

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

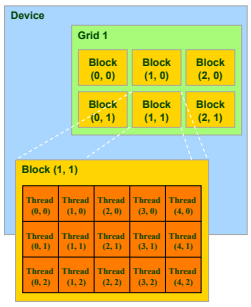
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

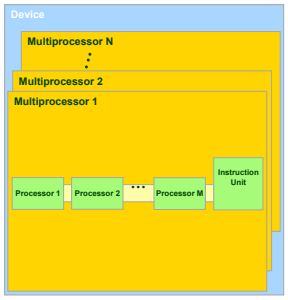
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

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

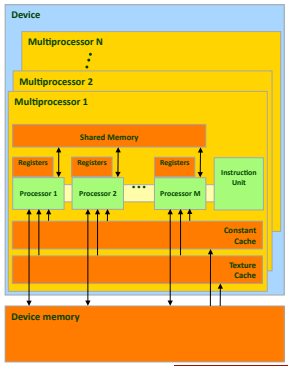


Hardware Execution Model

I. SIMD Execution of warpsize=M threads (from single block)
  – Result is a set of instruction streams roughly equal to # blocks in thread divided by warpsize

II. Multithreaded Execution across different instruction streams within block
  – Also possibly across different blocks if there are more blocks than SMs

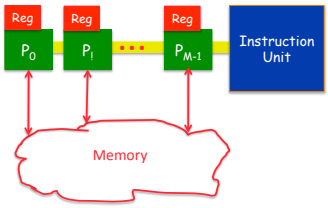
III. Each block mapped to single SM
  – No direct interaction across SMs



Example SIMD Execution

"Count 3" kernel function

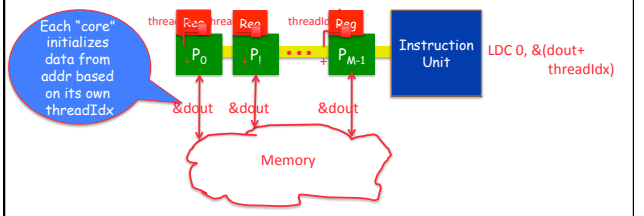
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 3);
}
```



Example SIMD Execution

"Count 3" kernel function

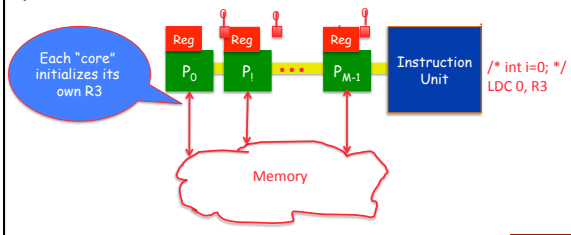
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 3);
}
```



Example SIMD Execution

"Count 3" kernel function

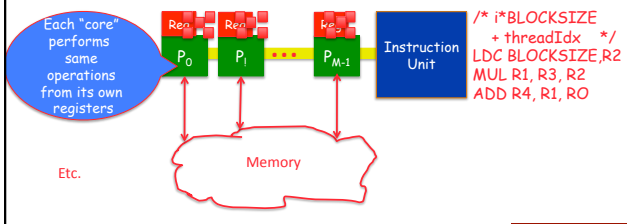
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 3);
}
```



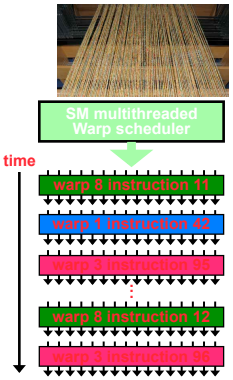
Example SIMD Execution

"Count 3" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 3);
}
```



SM Warp Scheduling

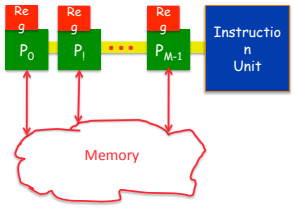


- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```

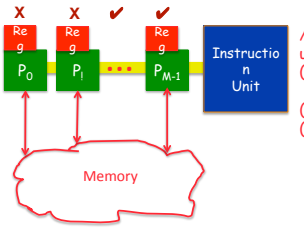


compare threadIdx,2

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx.x >= 2) {
   out[threadIdx.x] += 100;
}
else {
   out[threadIdx.x] += 10;
}
```

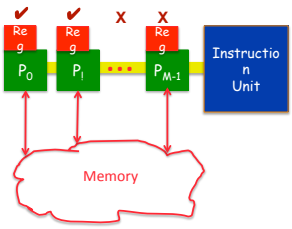


/* possibly predicated using CC */
(CC) LD R5, &(out+threadIdx.x)
(CC) ADD R5, R5, 100
(CC) ST R5, &(out+threadIdx.x)

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```



/* possibly predicated using CC */
(not CC) LD R5, &(out+threadIdx)
(not CC) ADD R5, R5, 10
(not CC) ST R5, &(out+threadIdx)

A Very Simple Execution Model

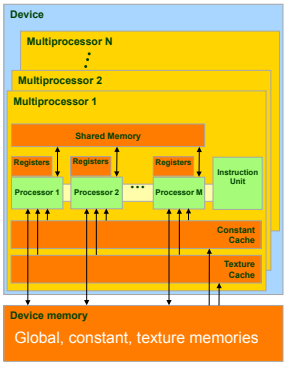
- No branch prediction
  - Just evaluate branch targets and wait for resolution
  - But wait is only a small number of cycles
- No speculation
  - Only execute useful instructions

Terminology

- Divergent paths
  - Different threads within a warp take different control flow paths within a kernel function
  - N divergent paths in a warp?
    - An N-way divergent warp is serially issued over the N different paths using a hardware stack and per-thread predication logic to only write back results from the threads taking each divergent path.
    - Performance decreases by about a factor of N

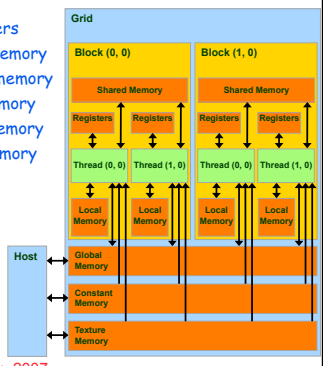
Hardware Implementation: Memory Architecture

- 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



Programmer's View: 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



Constant Memory Example

- Signal recognition:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute M_1V, M_2V, …, M_nV

```
__constant__ float d_signalVector[M];
__device__ float R[N][M];

__host__ void outerApplySignal () {
  float *h_inputSignal;
  dim3 dimGrid(N);
  dim3 dimBlock(M);
  cudaMemcpyToSymbol (d_signalVector,
      h_inputSignal, M*sizeof(float));
  ApplySignal<<<dimGrid,dimBlock>>>(M);
}
```

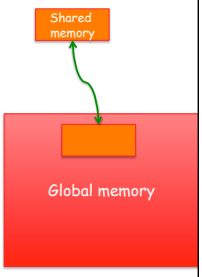
```
__global__ void ApplySignal (int M) {
  float result = 0.0; /* register */

  for (j=0; j<M; j++)
     result += d_M[blockIdx.x][threadIdx.x][j] *
           d_signalVector[j];
  R[blockIdx.x][threadIdx.x] = result;
}
```

Additional Detail

- Suppose each thread accesses different data from constant memory on same instruction
  - Reuse across threads?
    - Consider capacity of constant cache and locality
    - Code transformation needed? (later in lecture)
    - Cache latency proportional to number of accesses in a warp
  - No reuse?
    - Should not be in constant memory.

Now Let's Look at Shared Memory

- Common Programming Pattern (5.1.2 of CUDA manual)
  - Load data into shared memory
  - Synchronize (if necessary)
  - Operate on data in shared memory
  - Synchronize (if necessary)
  - Write intermediate results to global memory
  - Repeat until done



Familiar concept?

Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```
extern __shared__ float d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block */
/* shared memory*/
__host__ void outerCompute() {
  compute<<<gs,bs,MEMSIZE>>>();
}
__global__ void compute() {
   d_s_array[i] = …;
}
```

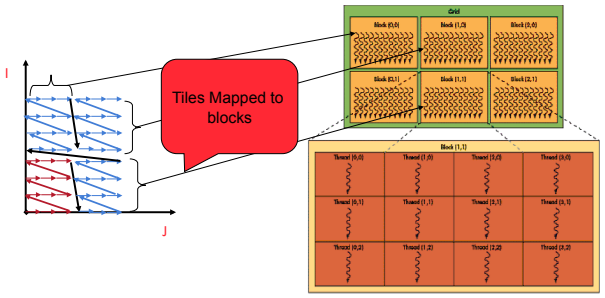
```
__global__ void compute2() {
  __shared__ float d_s_array[M];

  /* create or copy from global memory */
  d_s_array[j] = …;

 /* write result back to global memory */
  d_g_array[j] = d_s_array[j];
}
```

Using Loop Transformations in GPU Code Generation

Cod transformations	Application	
	Conventional Architectures	GPU
Tiling	•Manage reuse in limited storage	•Manage reuse in limited storage •Partition parallel execution at 2 levels
Data-copy	•Eliminate conflict misses in cache	•Copy data to specialized memory structures
Permutation	•Reorder loop structure to enable other optimizations	•Reorder loop structure to enable other optimizations
Unrolling	•Exposes fine-grain parallelism by replicating the loop body	•Exposes fine-grain parallelism by replicating the loop body
•••		

THE UNIVERSITY OF UTAH

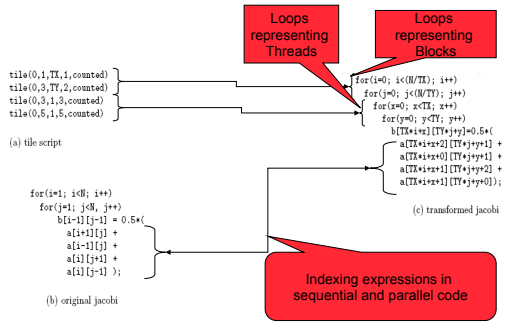
Tiling for Computation Mapping in GPUs



2D blocks and threads per block arrangement

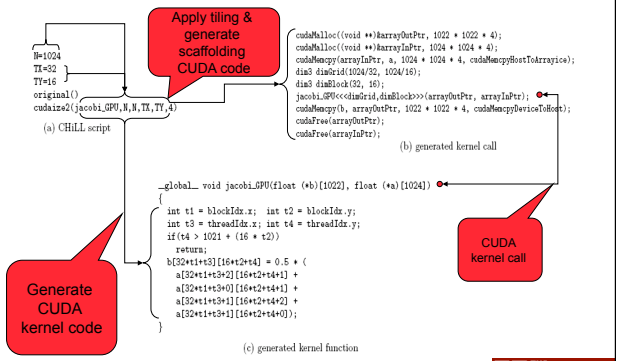
THE UNIVERSITY OF UTAH

Code Optimization and Generation: Jacobi Relaxation



THE UNIVERSITY OF UTAH

Code Optimization and Generation

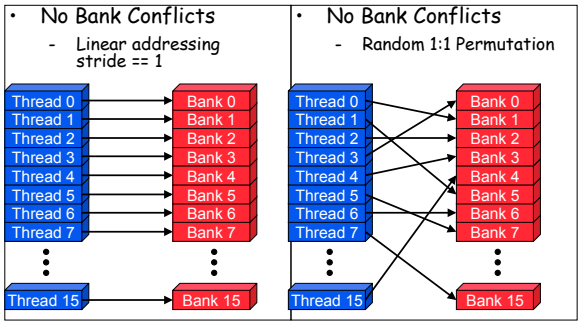


THE UNIVERSITY OF UTAH

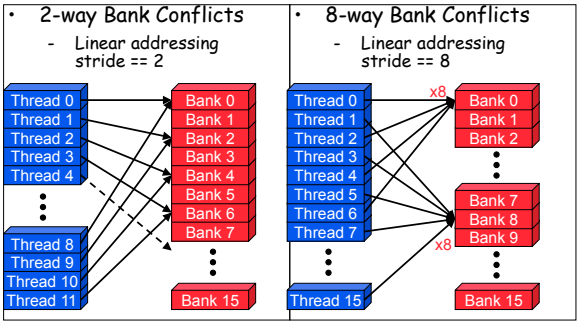
Bandwidth to Shared Memory: Parallel Memory Accesses

- Consider each thread accessing a different location in shared memory
- Bandwidth maximized if each one is able to proceed ***in parallel***
- Hardware to support this
  - ***Banked memory:*** each bank can support an access on every memory cycle

Bank Addressing Examples



Bank Addressing Examples



How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

THE UNIVERSITY OF UTAH

Global Memory Accesses

- Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities
- Given an address to load or store, memory returns/ updates "segments" of either 32 bytes, 64 bytes or 128 bytes
- Maximizing bandwidth:
  - Operate on an ***entire*** 128 byte segment for each memory transfer

THE UNIVERSITY OF UTAH

Understanding Global Memory Accesses

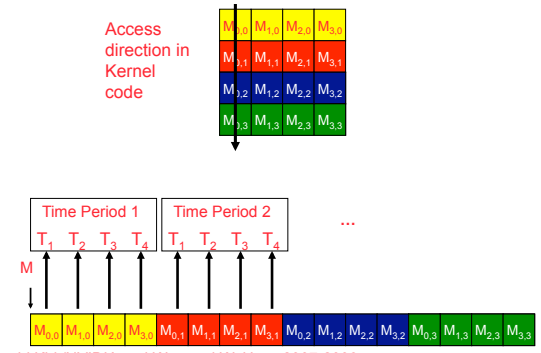
Memory protocol for compute capability 1.2* (CUDA Manual 5.1.2.1)

- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads ***in half-warp*** are serviced

*Includes Tesla and GTX platforms

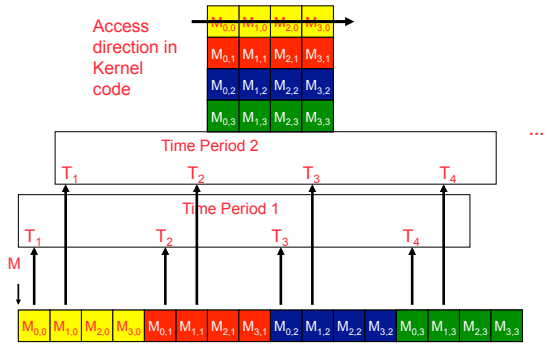
THE UNIVERSITY OF UTAH

Memory Layout of a Matrix in C



THE UNIVERSITY OF UTAH

Memory Layout of a Matrix in C



THE UNIVERSITY OF UTAH

Summary of Lecture

- A deeper probe of performance issues
  - Execution model
  - Control flow
  - Heterogeneous memory hierarchy
    - Locality and bandwidth
  - Tiling for CUDA code generation

THE UNIVERSITY OF UTAH