L17: CUDA, cont.

October 28, 2010

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)
  - 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 Wednesday, November 3

- 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
### From L1: The fastest computer in the world today

- **What is its name?** Jaguar (Cray XT5)
- **Where is it located?** Oak Ridge National Laboratory
- **How many processors does it have?** ~37,000 processor chips (224,162 cores)
- **What kind of processors?** AMD 6-core Opterons
- **How fast is it?** 1.759 Petaflop/second

One quadrillion operations/s

1 x 10\(^{16}\)

See [http://www.top500.org](http://www.top500.org)

---

### From L1: The 2nd fastest computer in the world

- **What is its name?** Jaguar (Cray XT5)
- **Where is it located?** Oak Ridge National Laboratory
- **How many processors does it have?** ~37,000 processor chips (224,162 cores)
- **What kind of processors?** AMD 6-core Opterons
- **How fast is it?** 1.759 Petaflop/second

One quadrillion operations/s

1 x 10\(^{16}\)

See [http://www.top500.org](http://www.top500.org)

---

### Breaking news: The fastest computer in the world

- **What is its name?** Tianhe-1A
- **Where is it located?** National Supercomputer Center in Tianjin
- **How many processors does it have?** ~21,000 processor chips (14,000 CPUs and 7,000 Tesla Fermis)
- **What kind of processors?** CPU? NVIDIA Fermi
- **How fast is it?** 2.507 Petaflop/second

2.5 quadrillion operations/s

1 x 10\(^{16}\)

See [http://www.top500.org](http://www.top500.org)

---

### 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)

[http://courses.ece.uiuc.edu/ece498/hw1/](http://courses.ece.uiuc.edu/ece498/hw1/)

and Austin Robison (NVIDIA)

11/05/09
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

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

```c
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);
}

Each "core" initializes data from address based on its own threadIdx

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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
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)

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 \( M_1 v, M_2 v, \ldots, M_n v \)

```c
__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;
    cudaMemcpyToSymbol (d_signalVector, h_inputSignal, M*sizeof(float));
    ApplySignal<<<dimGrid,dimBlock>>>(M);
}
```
**Additional Detail**

- Suppose each thread accesses different data from constant memory on some 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

**Mechanics of Using Shared Memory**

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

- Examples:
  ```c
  extern __shared__ float d_s_array[];
  __global__ void compute2()
  {
    __shared__ float d_s_array[M];
    // a form of dynamic allocation
    // MEMSIZE is size of per-block
    // shared memory
    __host__ void outerCompute()
    {
      compute<<<gs,bs,MEMSIZE>>>(
        __shared__ float d_s_array[]);
      __global__ void compute()
      {
        d_s_array[i] = ...;
        // create or copy from global memory
        d_g_array[i] = d_s_array[i];
      } // write result back to global memory
      d_g_array[i] = d_s_array[i];
    }
  }
  ```

**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

#### No Bank Conflicts
- Linear addressing
  - stride == 1

#### No Bank Conflicts
- Random 1:1 Permutation

### Bank Addressing Examples

#### 2-way Bank Conflicts
- Linear addressing
  - stride == 2

#### 8-way Bank Conflicts
- Linear addressing
  - stride == 8

---

### 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
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

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
Summary of Lecture

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