L14: CUDA, cont.
Execution Model and Memory
Hierarchy

October 27, 2011

Here's the code

.. Initialize th[i][j] = 0 ..
/* compute array convolution */
for(m = 0; m < IMAGE_NROWS - TEMPLATE_NROWS + 1; m++)
  for(n = 0; n < IMAGE_NCOLS - TEMPLATE_NCOLS + 1; n++)
    for(i = 0; i < TEMPLATE_NROWS; i++)
      for(j = 0; j < TEMPLATE_NCOLS; j++)
        if(mask[i][j] != 0)
          th[m+n] += image[i+m][j+n];
  
/* scale array with bright count and template bias */
.. th[i][j] = th[i][j] * bc - bias;

Programming Assignment 3. Due 11:59PM Nov. 7

• Purpose:
  - Synthesize the concepts you have learned so far
  - Data parallelism, locality and task parallelism
  - Image processing computation adapted from a real application

• Turn in using handin program on CADE machines
  - Handin cs4961 proj3 <file>
  - Include code + README

• Three Parts:
  1. Locality optimization (50%): Improve performance using
     locality optimizations only (no parallelism)
  2. Data parallelism (20%): Improve performance using locality
     optimizations plus data parallel constructs in OpenMP
  3. Task parallelism (30%): Code will not be faster by adding
     task parallelism, but you can improve time to first result.
     Use task parallelism in conjunction with data parallelism per
     my message from the previous assignment.

Things to think about

• Beyond the assigned work, how does parallelization
  affect the profitability of locality optimizations?

• What happens if you make the IMAGE SIZE larger
  (1024x1024 or even 2048x2048)?
  - You'll need to use "unlimit stacksize" to run these.

• What happens if you repeat this experiment on a
  different architecture with a different memory
  hierarchy (in particular, smaller L2 cache)?

• How does SSE or other multimedia extensions affect
  performance and optimization selection?
Reminder About Final Project

- **Purpose:**
  - A chance to dig deeper into a parallel programming model and explore concepts.
  - Research experience
  - Freedom to pick your own problem and solution, get feedback
  - Communication skills
  - Present results to communicate 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
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - MPI + OpenMP
  - MPI + SSE
  - MPI + CUDA
- **Present results in a poster session on the last day of class.**

Example Projects

- Look in the textbook or look on-line
  - Chapter 6: N-body, Tree search
  - Chapters 3 and 5: Sorting
  - Image and signal processing algorithms
  - Graphics algorithms
  - Stencil computations
  - FFT
  - Graph algorithms
- Other domains...
- **Must change it up in some way from text**
  - Different language/strategy

Details and Schedule

- 2-3 person projects
  - Let me know if you need help finding a team
- **Ok to combine with project for other class, but expectations will be higher and professors will discuss**
- Each group must talk to me about their project between now and November 10
  - Before/after class, during office hours or by appointment
  - Bring written description of project, slides are fine
  - Must include your plan and how the work will be shared across the team
- I must sign off by November 22 (in writing)
- Dry run on December 6
- Poster presentation on December 8

Strategy

- A lesson in research
  - Big vision is great, but make sure you have an evolutionary plan where success comes in stages
  - Sometimes even the best students get too ambitious and struggle
  - Parallel programming is hard
  - Some of you will pick problems that don’t speed up well and we’ll need to figure out what to do
  - There are many opportunities to recover if you have problems
    - I’ll check in with you a few times and redirect if needed
    - Feel free to ask for help
    - Optional final report can boost your grade, particularly if things are not working on the last day of classes
**More on Niagara**

- Target applications are server-class, business operations
- Characterization:
  - Floating point?
  - Array-based computation?
  - Support for VIS 2.0 SIMD instruction set
  - 64-way multithreading (8-way per processor, 8 processors)

---

**The 2nd 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
  \(1 \times 10^{18}\)

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)  
[see 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 Manual, particularly Chapters 2 and 4
  (download from nvidia.com/cudazone)
- Nice series from Dr. Dobbs Journal by Rob Farber
  (link on class website)
  - 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 warp$size \times M$ threads (from single block)
   - Result is a set of instruction streams roughly equal to $M$ blocks in thread divided by warp$size$

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 6" kernel function

```c
for (int i = 0; i < SIZE/BLOCKSIZE; i++) {
    int val = d_in[i*BLOCKSIZE + threadIdx.x];
    d_out[threadIdx.x] += compare(val, 76);
}
```
Example SIMD Execution

"Count 6" 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, 6);
}

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

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;
}

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]

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 (DRAM).
- 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.
  - Data cache (Fermi only).

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.

Targets of Memory Hierarchy Optimizations

- Reduce memory latency
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion.
- Maximize memory bandwidth
  - Bandwidth is the amount of useful data that can be retrieved over a time interval.
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain.

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

Memory Layout of a Matrix in C

Access direction in Kernel code

Shared memory

Global 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:
  ```c
  extern __shared__ float d_s_array[ ];  
  __global__ void compute() {
      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`
  - Random 1:1 Permutation

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

- 8-way Bank Conflicts
  - Linear addressing
    - `stride == 8`
How addresses map to banks on G80 (older technology)

- 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-warsps, 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

Summary of Lecture

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