L4: Hardware Execution Model and Overview

January 26, 2009

Administrative

- First assignment out, due Friday at 5PM
  - Any questions?
- New mailing list:
  - cs6963-discussion@list.eng.utah.edu
  - Please use for all questions suitable for the whole class
  - Feel free to answer your classmates questions!

Outline

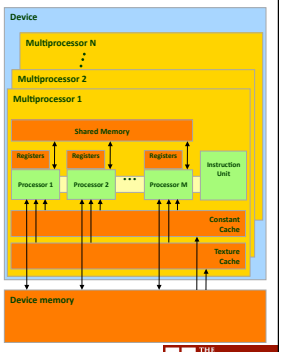
- Single Instruction Multiple Data (SIMD)
- Multithreading
- Scheduling instructions for SIMD, multithreaded multiprocessor
- How it comes together
- Reading:
  Ch 2.3 in Grama et al.

Recall 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

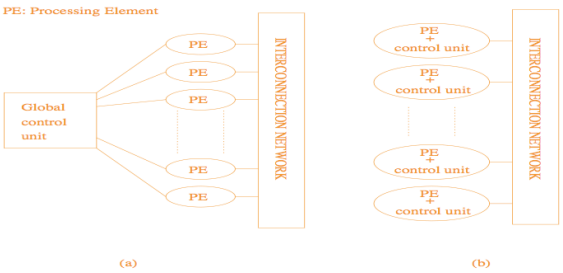
III. Each block mapped to single SM
  - No direct interaction across SMs



Predominant Control Mechanisms: Some definitions

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 95: **A[1:n] = A[1:n] + B[1:n]** Kernel fns w/in block: compute<<<gs,bs,msize>>>
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	OpenMP parallel loop: **forall (i=0; i<n; i++)** Kernel fns across blocks compute<<<gs,bs,msize>>>
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: **if ($threadIdx == 0) { }**

SIMD vs. MIMD Processors



A typical SIMD architecture (a) and a typical MIMD architecture (b).

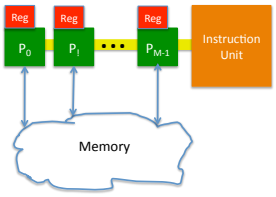
I. SIMD

- Motivation:
  - Data-parallel computations map well to architectures that apply the same computation repeatedly to different data
  - Conserve control units and simplify coordination
- Analogy to light switch

Example SIMD Execution

"Count 6" kernel function

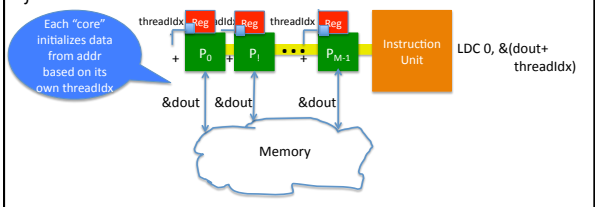
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 6);
}
```



Example SIMD Execution

"Count 6" kernel function

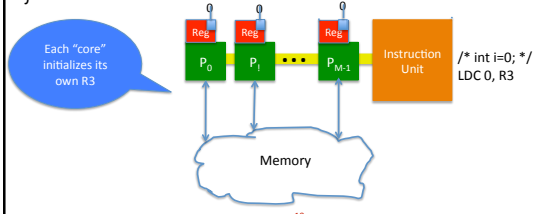
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 6);
}
```



Example SIMD Execution

"Count 6" kernel function

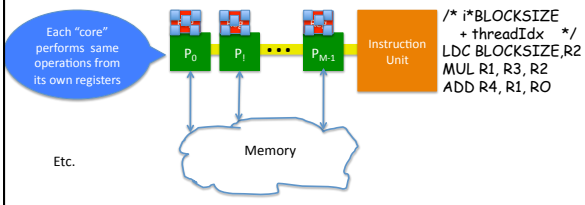
```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
  int val = d_in[i*BLOCKSIZE + threadIdx.x];
  d_out[threadIdx.x] += compare(val, 6);
}
```



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);
}
```

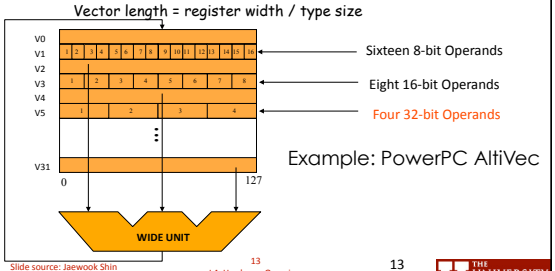


Overview of SIMD Programming

- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE-3
  - Graphics and games processors (example, IBM Cell)
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model?
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!) or code is automatically generated by a compiler
  - GPUs are programmed by games developers (domain-specific)
  - Accelerators typically use their own proprietary tools

Aside: Multimedia Extensions like SSE-3

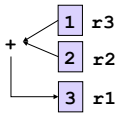
- COMPLETELY DIFFERENT ARCHITECTURE!
- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
    Vector length = register width / type size



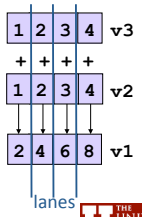
Example: PowerPC AltiVec

Aside: Multimedia Extensions Scalar vs. SIMD Operation

```
Scalar: add r1,r2,r3
```




```
SIMD: vadd<sws> v1,v2,v3
```



II. Multithreading: Motivation

- Each arithmetic instruction includes the following sequence

Activity	Cost	Note
Load operands	As much as O(100) cycles	Depends on location
Compute	O(1) cycles	Accesses registers
Store result	As much as O(100) cycles	Depends on location

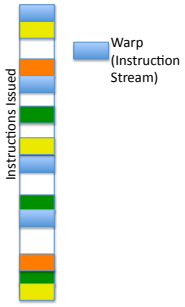
- ***Memory latency,*** the time in cycles to access memory, limits utilization of compute engines

Thread-Level Parallelism

- Motivation:
  - a single thread leaves a processor under-utilized for most of the time
  - by doubling processor area, single thread performance barely improves
- Strategies for thread-level parallelism:
  - multiple threads share the same large processor reduces under-utilization, efficient resource allocation
    Multi-Threading
  - each thread executes on its own mini processor simple design, low interference between threads
    Multi-Processing

What Resources are Shared?

- Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)
- For correctness, each thread needs its own program counter (PC), and its own logical regs (on this hardware, each gets its own physical regs)
- Functional units, instruction unit, i-cache shared by all threads

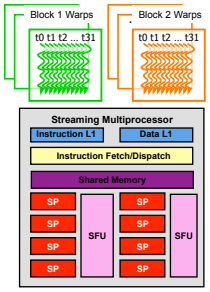


Aside: Multithreading

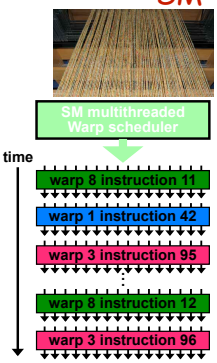
- Historically, supercomputers targeting non-numeric computation
  - HEP, Tera MTA
- Now common in commodity microprocessors
  - Simultaneous multithreading:
    - Multiple threads may come from different streams, can issue from multiple streams in single issue slot
    - Alpha 21464 and Pentium 4 are examples
- CUDA somewhat simplified:
  - A full warp scheduled at a time

Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



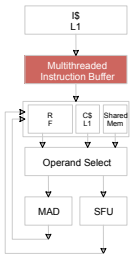
SM Warp Scheduling



- 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

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand ***scoreboarding*** used to prevent hazards
- Issue selection based on round-robin/ age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp

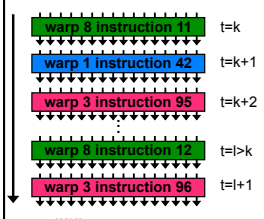


Scoreboarding

- How to determine if a thread is ready to execute?
- A ***scoreboard*** is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

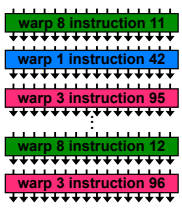


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
…		

Schedule at time k

Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

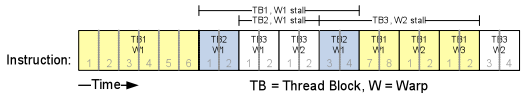


Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

Schedule at time k+1

Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

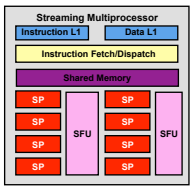


III. How it Comes Together

- Each block mapped to different SM
- If #blocks in a grid exceeds number of SMs,
  - multiple blocks mapped to an SM
  - treated independently
  - provides more warps to scheduler so good as long as resources not exceeded
- Within a block, threads observe SIMD model, and synchronize using __syncthreads()
- Across blocks, interaction through global memory

Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and memory access



Summary of Lecture

- SIMD execution model within a warp, and conceptually within a block
- MIMD execution model across blocks
- Multithreading of SMs used to hide memory latency
  - Motivation for lots of threads to be concurrently active
- Scoreboarding used to track warps ready to execute

What's Coming

- Next time:
  - Correctness of parallelization (deferred from today)
- Next week:
  - Managing the shared memories
- February 18:
  - Presentation on MPM by Jim Guilkey