GPU - accelerated systems
Tutorial 1

slides by
Mark Silberstein
Principles and terminology of parallel programming
Parallelizing Game of Life (Cellular automaton)

- Given a 2D grid
- $v^t(i,j) = F(v^{t-1}(\text{of all its neighbors}))$
Problem partitioning

- Domain decomposition
  - (SPMD)
  - Input domain
  - Output domain
  - Both

- Functional decomposition
  - (MPMD)
  - Independent tasks
  - Pipelining
We choose: domain decomposition

- The field is split between processors
Issue 1. Memory

- Can we access $v(i+1,j)$ from CPU 0 as in serial program?
It depends...

- **YES**: Shared memory space architecture: same memory space
  
  ![Diagram 1: Shared Memory Space](image)

- **NO**: Distributed memory space architecture: disjoint memory space
  
  ![Diagram 2: Distributed Memory Space](image)
Memory architecture

- CPU0: Time to access $v(i+1,j)$ = Time to access $v(i-1,j)$?
Hardware shared memory flavors 1

- Uniform memory access: **UMA**
  - Same cost of accessing **any** data by all processors
Hardware shared memory flavors 2

- NON-Uniform memory access: **NUMA**
Memory-optimized programming

- Most modern systems are NUMA or distributed
- Access time difference: local vs. remote data: x100-10000
- Memory accesses: main source of optimization in parallel and distributed programs
Issue 2: **Control**

- Can we assign one \( \nu \) per CPU?
Task management overhead

- Each task has a state that should be managed
- More tasks – more state to manage
  - Who manages tasks?
- How many tasks should be run?
  - Does that depend on $F$?
    - Reminder: $v(i,j) = F(\text{all } v's \text{ neighbors})$
Question

- Every process reads the data from its neighbors
- Will it produce correct results?
Synchronization

- The order of reads and writes made in different tasks is non-deterministic
- Synchronization required to enforce the order
  - Locks, semaphores, barriers, conditionals
Check point

Fundamental hardware-related issues

- Memory accesses
  - Optimizing locality of accesses
- Control
  - Overhead
- Synchronization
Parallel programming issues

- We decide to split this 3x3 grid like this:

Problems?
Issue 1: Load balancing

- Always waiting for the slowest task
- Solutions?
Issue 2: Granularity

- G = Computation/Communication

- Fine-grain parallelism.
  - G is small
  - Good load balancing
  - *Potentially* high overhead

- Coarse-grain parallelism
  - G is large
  - *Potentially* bad load balancing
  - Low overhead

Which granularity works for you?
It depends..

• For each combination of computing platform and parallel application
  - *The goal is to minimize overheads and maximize utilization*

• High performance requires
  - Enough parallelism to keep ALUs busy
  - Low *relative* overheads (communications, synchronization, task control, memory accesses versus computations)
  - Good load balancing
Summary so far

- Parallelism does not come for free
  - Overhead
  - Memory-aware access
  - Synchronization
  - Granularity
Performance of parallel programs

• A parallel system is a combination of a parallel algorithm and an underlying platform

• Intuitive measures
  – Wall clock time
  – Speedup = (Serial time)/(Parallel time)
  – GFLOPs/s = how well the hardware is exploited

• Need more:
  – Scalability: Speedup as a function of #CPUs
  – Efficiency: Speedup/#CPUs
Question

• If I use 2 processors, will the program run twice as fast?
Sources of inefficiency

- Parallel program may need more work to do
- Communication overhead
- Idling
  - Load imbalance
  - Serial part (insufficient parallelism)
- Memory accesses
- Control
- Wrong choice of granularity for a given platform
Upper bound on speedup
Amdahl's law

- Sequential component **limits the speedup**
- Split program into
  - *ideally parallelizable*: \( \%A \) (fraction parallel):
  - *totally sequential*: 1-A

\[
\text{Scalability} = \text{Speedup}\left(\text{#CPUs}\right) = \frac{1}{A/\text{#CPUs} + (1-A)}
\]
Bad news

- So why do we need machines with 1000x CPUs?

Source: wikipedia
Strong scaling/weak scaling

• Strong scaling (Amdahl's law)
  – problem size is fixed: *solve same problems faster*

• Weak scaling
  – Problem size grows: *make larger problems feasible*
  – Keep the amount of work per CPU when adding more CPUs so as to reduce 1-G proportionally
Amdahl's law = Common sense
Your code is as fast as your bottleneck

• Always use common sense when
  – Optimizing code
  – Offloading to a GPU

• Ex: Read data from disk and multiply x2 every element. Offload to GPU?
Summary so far

- Parallel performance is a subtle matter
  - Need to understand the hardware
  - Need to compare to good serial program

- **Amdahl's law:** understand the assumptions!!!!
  - Too pessimistic
    - Fraction parallel independent of #CPUs
    - Assumes fixed problem size
    - Ignores caches
  - Too optimistic
    - Assumes perfect load balancing, no idling, equal CPUs speeds
Memory

• Performance characteristics

Example: which drop will reach the other side first?
Memory

• Performance characteristics

• Latency – time between data request and receipt
• Bandwidth – rate at which data is moved
Why latency is a problem

• Processor
  – 1 floating point ADD = 1 cycle (1ns)
  – Memory access latency=100ns.
  – Memory bandwidth = Infinity
  – Assume no cache

• RAW ALU performance: 1GFLOPs/s

• Observed performance for adding two vectors?
Why latency is a problem

- Observed performance for adding two vectors?
  - We need 3 memory accesses = 300ns per 1 addition => 300 times slower than ALU
Why bandwidth is a problem

• Processor
  – Memory latency = 0 ns
  – Memory bandwidth = 1GB/s
  – 1 floating point ADD = 1 cycle (1ns)

• RAW ALU performance: 1GFLOPs/s

• Observed performance for adding two vectors?
Why bandwidth is a problem

- Observed performance for adding two vectors?
  
  We need 12 bytes of memory access per one cycle: 12 GB/s. But we can do only 1GB/s => 12 times slower