Data Parallel Programming II

CS149
Lecture 10

Announcements

• PS1 due today

Lecture Overview

• More details on data parallel programming
• Implementing vector instructions
• More on general SIMD
  - Programming
  - Implementation
• Implementation of stream programs

Review

forall (i = 1..N)

Instances:
...

Vector Hardware

Vector registers

FP Unit

Review

A[1..N] = A[1..N] + 1;

• A vector instruction works on an unbounded size argument array (or arrays)
• The vector registers have fixed size
• The program ignores the resource constraint
  - But to run this code we have to deal with fixed size registers
Strip Mining

- Idea: Replace vector instructions by a loop
- Loop body is a vector instruction operating on chunks of the argument arrays equal in size to the vector registers
- The loop iterates through all such chunks of the array

Strip Mining Example

$$A[1..N] = A[1..N] + 1;$$

... becomes ...

for i = 1 to N-1 by k
$$A[i..i+k-1] = A[1..i+k-1] + 1;$$

where k is the size of the vector registers (assuming $N \mod k = 0$)

Higher-Dimensional Strip Mining

- Most often only strip mine in one dimension
  - But can strip mine in two or more dimensions
  $$A[1..N,1..M] += 1$$

Given register size k where $i \times j = k$

for $x = 1$ to $N-1$ by $i$
for $y = 1$ to $M-1$ by $j$
$$A[ x..x+i-1, y..y+j-1 ] += 1$$

Assembly Code

for i = 1 to N-1 by k
$$A[i..i+k-1] = A[1..i+k-1] + 1;$$
... compiles to ...

la r1, A
load_veci v2, 1   // load v2 with all 1’s
top: load_vec v1, r1
add_vec v3, v1, v2
store_vec r1,v3
addi r1, r1, k*size
(test&branch top)

Comments

- The bulk load is efficient
  - Accessing a contiguous block of k words faster than access to k randomly distributed words
- The vector operation is efficient
  - Deeply pipelined, fine-grain overlap of all k operations
- But the code overall is not very efficient

Software Pipelining

la r1, A
load_veci v4, 1    // load v2 with all 1’s
top: // invariant: r1 points to already computed block
load_vec v1,r1-2k*size // load data for next block
add_vec v3, v2, v4    // do op for current block
store_vec r1,v5   // store previous block
mv v2, v1    // or rename registers
mv v5, v3
addi r1,k*size
(test&branch top)
... drain ...
Discussion

- Get all machine resources running in parallel
  - Not just processors, but memory/network
- Overlap computation & communication
  - Straightforward for a compiler
  - Tedious for a programmer
  - Generally requires bookkeeping for old results to be store, results to be computed, next arguments to fetch

Summary

- Even for a simple, single vector instruction
- Important to
  - Block data to size of vector registers
  - Reorder/software pipeline instructions to overlap communication & computation
- Mechanical but machine dependent
  - Let a compiler do this!

Current Compilers

- Vector processing was a big deal in '80’s
  - And vector support in compilers was good
- Has since atrophied
  - Not many production compilers do a good job vectorizing today
    - E.g., gcc
    - Even though hardware provides vector instructions
- Bottom line
  - Programmers currently need to worry about this

Opteron Example

- Sum 8 unsigned short ints
- Fastest instruction sequence:
  - `psubb %mm0, %mm0`
  - `psadbw &a[i], %mm0`
  - `movd %mm0, sum`

Current Machines

- CPUs generally provide vector units
- Size of vector registers has varied
  - 2 - 64 words is the historical range
  - Depends on anticipated apps
    - Want compute - communication balance
    - Need lots of data parallelism to make 64 element vector registers pay off!
- Current sweet spot seems to be 8-16
Review: General Data-Parallelism

- Beyond vector programming, some languages provide general maps/folds on arrays/sequences
  - APL, FP, FL, map-reduce
- And one other operation: parallel prefix

Parallel Prefix Example

- Given a sequence of integers
  - < 7, 9, 1, 0, 2, 5, 6 >
- Compute all sums of prefixes
  - < 7, 16, 17, 17, 19, 21, 26, 32 >

Sequential Code

```
for (i = 1; i < n; i++)
    A[i] += A[i-1]
```

• This appears to be inherently sequential …

Model: One Processor per Element

Time Step

Processor

Notes

- Some of the processors have partial sums
  - Odd processor i = A[i]
- Most of the time processors are idle
  - log n time steps
  - n total operations
Parallel Prefix

\[
\begin{array}{c}
\text{for } i = 1 \text{ to } \log n \\
\text{do all } k \text{ in parallel do} \\
\text{if } k \geq 2^i \text{ then} \\
A[k] := A[k - 2^{i-1}] + A[k]
\end{array}
\]

Discussion

- Defined for any binary associative operator
  - +, *, and, or, matrix multiply ...
- Work
  - \(O(n)\) computations per time step
  - \(\log n\) time steps
- Compare with sequential algorithm!

Regular Expression Matching

- Regular expressions = finite automata
- Finite automata = table \(T\)
  - \(T: \text{State} \times \text{Input} \rightarrow \text{State}\)

Example

- Example: A string of 1's & 0's with an odd number of 1's

Regular Expression Matching in Parallel

- Given a string we want to
  - Assign a state to each input character such that
  - The first character is an initial state
  - The last character is a final state
  - Adjacent characters satisfy the transition relation
Insight

• Observation
  - Each character defines a function states -> states
  - i.e., a column of the transition table

• Each sequence of characters also defines a function states -> states
  - What state the automaton ends up in given an initial state
  - And this composition operator is associative

Algorithm

• Replace each character in the input by is transition relation

• Perform parallel-prefix
  - Combining operation is composition of transition relations

• Select the start state from resulting transition relations
  - This is the state of the automaton

Example

Implementing General D.P.

• We want to implement maps, folds, and parallel prefix operations over sequences

• In general, |sequence| >> # of processors

• For sequence of length N with P processors
  - Give each processor N/P adjacent elements
  - Compute results on each block locally
  - Then combine using parallel operations

An Issue

• This works fine for map and fold
  - Serial & parallel versions do the same work

• Not so obvious for parallel prefix
  - Simulating parallel version on 1 processor may do extra work . . .

Two Solutions

• Solution 1
  - Do serial version of prefix operation locally
  - Do parallel phase
  - Do a local serial phase to fold in prefix result

• Solution 2
  - Do serial version of prefix operation locally
  - Do parallel phase
  - Simply remember prefix result and fold in on next operation
A Word on Streaming Languages

• Recall
  kernel void f(float arg<>, out float res<>, int x) {
    res = arg + x;  }

• Reality
  kernel void f(float arg<>, out float res<>, int x) {
    arg = pop(arg);
    push(res, arg + x);
  }

Compiling Streaming Languages

• Language requires gets/stores to the streams to be syntactically explicit

• Compiler can count data read & written
  - Compile similarly to vector instructions
  - Prefetch next batch of data
  - Compute on current batch of data
  - Store results from previous data

• Pop/push operations guarantee good locality