Data-Parallel Programming

CS149
Lecture 9

Announcements

- PA2 due today
- PS1 due Feb 4

Back to Basics

- Sequential
  for (i = 1; i < N; i++)
  A[i] = B[i] + C[i];

- Parallel
  forall (i = 1..N)
  A[i] = B[i] + C[i];

Vectorization

forall (i = 1..N)
A[i] = B[i] + C[i];

- Vector form
  A[1..N] = B[1..N] + C[1..N]

Vector Statements

- A single statement
- Where all instances of the statement are independent
- Generally one operation on the RHS

Compare

forall (i = 1..N) {
  A[i] = B[i] + C[i];
  B[i] = D[i] * E[i];
}

A[1..N] = B[1..N] + C[1..N]
B[1..N] = D[1..N] * E[1..N]
**Vector Programming**

- In the '70s and '80s, vectorization was king
  - The main route to high performance processing
- Efforts to build automatically parallelizing compilers
  - Convert sequential loops to vector instructions
- Eventually gave way to vector extensions
  - Added vector notation to C and FORTRAN

**Discussion**

\[ A[1..N] = B[1..N] + C[1..N] \]

- Single instruction (operation)
- Done on multiple data items
- Recall: SIMD
- Data-parallel programming
  - Vector instructions are an example of data parallelism

**Discussion**

\[ A[1..N] = B[1..N] + C[1..N] \]
\[ B[1..N] = D[1..N] \times E[1..N] \]

- Bounded iteration
  - # of instances of a statement X unaffected by X
- One control context
  - Same operation done many times in parallel
- Synchronization after every statement
  - Instances of one statement are independent

**Example 1**

```c
for (i = 1; i < n; i++) {
    t = A[i] \times B[i];
    C[i] = t \times t;
}
```

**Example 2**

```c
for (i = 1; i < n; i++) {
    if (C[i] > 0) {
        A[i] += B[i]
    } else {
        A[i] += D[i]
    }
}
Example 3

\[
\text{for } (i = 1; i < n; i++) \\
\text{for}(j = 1; j < m; j++) \\
A[i] = 2 \times A[i-1];
\]

Example 4

\[
\text{for } (i = 1; i < n; i++) \\
\text{for}(j = 1; j < m; j++) \\
\]

Iteration Space

\[
\text{for } (i = 1; i < n; i++) \\
\text{for}(j = 1; j < m; j++)
\]

- Every loop nest defines an iteration space
  - Each point is one valuation of the index variables
  - Two nested loops = 2-dimensional space
- Loop transformations
  - Rearrange the iteration space
  - Linear transformations

More on Predication

- Guarded commands
  \[B[1..N]: A[1..N] = C[1..N];\]
- Other notations:
  \[A[1..N] = \text{vec\_select}(A[1..N],C[1..N],B[1..N]);\]
- Possible optimization:
  \[\text{if (vec\_any\_ne(B[1..N],vec(0,N)))} \\
  A[1..N] = \text{vec\_select}(A[1..N],C[1..N],B[1..N]);\]

Vectorization Techniques

- Scalar expansion
  - Temporaries in sequential code become arrays
  - Space implications!
- Predicated execution
  - Or guarded commands
  - Divergence wastes compute resources
- Loop rearrangements
  - Mostly commonly interchanging loops

Reductions

- Vector languages also provide reductions
  - Often built-in, but sometimes more general
  \[\text{sum} = \text{fold}(+, 0, A[1..N])\]
  Or
  \[\text{sum} = \text{vector\_sum}(A[1..N])\]
Map-Reduce vs. Vectorization

• This looks a lot like map-reduce . . .

• ... but it is more accurate to say map-reduce looks like a vector language

• Differences
  - Fine vs. coarse-grain parallelism
  - Map-reduce wires in <key,value> pairs as data type

Beyond Arrays

• Maps and folds make sense on other data types besides arrays
  - Sets
  - unordered collections
  - Lists
  - Ordered collections w/o layout constraint of arrays
  - Sequences
  - Like arrays

Beyond Vectorization

• Reductions on arbitrary datatypes
• Nested parallelism
• Streaming languages

Reductions on Arbitrary Datatypes

• Consider a list
  - [1, 2, 3, 4, 5]
• Each list element is a record of two elements
  - A head
  - A tail

Lists

```
null
: 1 : 2 : 3
```

```
null
: 1 : 2 : 3
```

```haskell
cons(1, cons(2, cons(3, null))
```
Reduction on Lists

fold(+,0,cons(1, cons(2, cons(3, nil))))

The General Scheme

• Consider a data type with \( N \) constructors
• Want a reduction producing type \( T \)
• For each constructor \( C(X_1, \ldots, X_n) \)
  - Supply a function from \( T^n \to T \)
• Example: Lists
  - Binary constructor \( \text{cons} \)
  - Nullary constructor \( \text{nil} \)

Works for Any (Tree) Datatype

• Consider
  - \( \text{TreeInt} ::= \text{Leaf(Int)} \mid \text{TreeInt} \times \text{TreeInt} \)
• A fold operation \( \text{TreeInt} \to T \) requires
  - A unary operation \( \text{Int} \to T \)
  - A binary operation \( T \times T \to T \)
• Example
  - \( \text{TreeIntFold(Id, +)} \)

Examples

• Count the number of nodes in a \( \text{TreeInt} \)
• Swap left and right branches in a \( \text{TreeInt} \)

Nested Parallelism

• Back to sequences
  - \( [1,2,3,4,5] \)
• A map-like operator
  - \( \{ F(x) : x \in S \} \)
• Example
  - \( \{ \text{sum(a)} : a \in [[1,2],[3,4,5],[6,7,8,9]] \} \)
### Some Operations on Sequences
- \#S: length of a sequence
- S[i]: indexing into a sequence
- S1 ++ S2: concatenate sequences

### Quicksort in NESL
```nesl
function Quicksort(S) =
  if (#S <= 1) then S
  else
    let a = S[rand(#S)];
    S1 = \{ x in S | x < a \};
    S2 = \{ x in S | e == a \};
    S3 = \{ x in S | x > a \};
    R = (Quicksort(v) : v in [S1, S3]);
    in R[0] ++ S2 ++ R[1]
```

### Streaming Languages
- Streaming languages emphasize:
  - Data parallel operations on sequences (streams)
  - Locality

- Numerous streaming languages today:
  - Mostly as research projects, but big influence on practice (e.g., CUDA)

### Language Design
- Streaming languages defined as extensions of an imperative base language:
  - The style in which these languages have been done
  - Inherit all features of base language:
    - E.g., C
    - Add streams:
      - new datatype like a sequence
    - Add kernels:
      - Special functions

### Kernels
```nesl
kernel void f(float arg<>, out float res<>, int x) {
  res = arg + x;
}
```

### Locality
- Note the emphasis on locality:
  - Streams cannot refer to global, static storage
  - Stream arguments are managed by the system

- An example of a local address space language:
  - All functional languages are local address space
  - But streaming languages are more extreme:
    - E.g., not possible to define a permute operation on a stream
Summary of Data Parallelism

- Single thread of control

- Well-specified synchronization
  - Parallel within a statement
  - Sequential across statements

- Surprising number of algorithms expressible
  - Reduction is a general concept!

- But number of efficient algorithms is smaller
  - Locality still an issue