Announcements:

- HW4
- ICES

Today:
- expression trees
- parallel
- poly
Expression Trees and Term Rewriting

Demos:

- Demo: lang/01 Expression Trees
- Demo: lang/02 Traversing Trees
- Demo: lang/03 Defining Custom Node Types
- Demo: lang/04 Common Operations

How do expression trees come to be? (not our problem here)
Embedded languages

Main challenge: Obtaining a syntax tree. Approaches?
Macros: Goals and Approaches

What is a macro?

What data do macro systems operate on?
Macros: Textual and Syntactic, **Hygiene**

Macros: What can go wrong if you’re not careful?

```c
#define INCI(i) do { int a=0; ++i; } while(0)

int main(void)
{
    int a = 4, b = 8;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

How can the problem above be avoided?
Towards Execution

Demo: lang/06 Towards Execution
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  Expression Trees
  Parallel Patterns and Array Languages

Polyhedral Representation and Transformation
Reduction

\[ y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N) \]

where \( N \) is the input size.

Also known as

- Lisp/Python function reduce (Scheme: fold)
- C++ STL `std::accumulate`
Reduction: Graph
Approach to Reduction

Can we do better?

“Tree” very imbalanced. What property of $f$ would allow ‘rebalancing’?

$$f(f(x, y), z) = f(x, f(y, z))$$

Looks less improbable if we let $x \circ y = f(x, y)$:

$$x \circ (y \circ z)) = (x \circ y) \circ z$$

Has a very familiar name: **Associativity**
Reduction: A Better Graph

Processor allocation?
Mapping Reduction to SIMD/GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.

*In particular:* Use multiple grid invocations to achieve inter-workgroup synchronization.
Map-Reduce

Sounds like this:

\[ y = f(\cdots f(f(g(x_1), g(x_2)),
\hspace{1cm}
g(x_3)), \ldots, g(x_N)) \]

where \( N \) is the input size.

- Lisp naming, again
- Mild generalization of reduction
Map-Reduce: Graph
\[ y_1 = x_1 \]
\[ y_2 = f(y_1, x_2) \]
\[ \vdots \]
\[ y_N = f(y_{N-1}, x_N) \]

where \( N \) is the input size. (Think: \( N \) large, \( f(x, y) = x + y \))

- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan
Again: Need assumptions on $f$. Associativity, commutativity.
Scan: Implementation
Scan: Implementation II

Two sweeps: Upward, downward, both tree-shape

On upward sweep:
- Get values $L$ and $R$ from left and right child
- Save $L$ in local variable $\text{Mine}$
- Compute $\text{Tmp} = L + R$ and pass to parent

On downward sweep:
- Get value $\text{Tmp}$ from parent
- Send $\text{Tmp}$ to left child
- Sent $\text{Tmp} + \text{Mine}$ to right child
Scan: Examples

Name examples of Prefix Sums/Scans:

Radix sort

Merlot
Data-parallel language: Goals

**Goal:** Design a full data-parallel programming language

**Example:** What should the (asymptotic) execution time for Quicksort be?

Question: What parallel primitive could be used to realize this?

Belloch ‘95
NESL Example: String Search

teststr = "string strap asop string" : [char]
>>> candidates = [0:#teststr-5];
candidates = [0, 1, 2, 3, .... : [int]
>>> {a == ‘s: a in teststr -> candidates};
>>> candidates = {c in candidates;
... a in teststr -> candidates | a == ‘s};
candidates = [0, 7, 13, 20, 24] : [int]
>>> candidates = {c in candidates;
... a in teststr -> {candidates+1:candidates}
... | a == ‘t};

▶ Work and depth of this example?
▶ NESL specifies work and depth for its constructs
▶ How can scans be used to realize this?

Blelloch ‘95
Array Languages

Idea:
▶ Operate on entire array at once
▶ Inherently data-parallel

Examples:
▶ APL, numpy
▶ Tensorflow (talk on Friday), Pytorch

Important axes of distinction:
▶ Lazy or eager
▶ Imperative (with in-place modification) or pure/functional
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Polyhedral Representation and Transformation
  Polyhedral Model: What?
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Polyhedral Representation and Transformation
  Polyhedral Model: What?
Basic Object: Presburger Set

Think of the problem statement here as representing an arbitrary-size (e.g.: dependency) graph.

*Presburger sets* correspond to a subset of predicate logic acting on tuples of integers.

**Important:** Think of this as a mathematical tool that can be used in many settings.
Basic Object: Presburger Set

Terms:

- Variables, Integer Constants
- +, −
- $\lfloor \cdot / d \rfloor$

Predicates:

- $(\text{Term}) \leq (\text{Term})$
- $(\text{Pred}) \land (\text{Pred})$, $(\text{Pred}) \lor (\text{Pred})$, ¬(Pred)
- $\exists v : (\text{Pred})(v)$

Sets: integer tuples for which a predicate is true

Verdoolaege ‘13
Presburger Sets: Reasoning

What’s “missing”? Why?

\[ \mathcal{P} \cdot \mathcal{P} + \mathcal{P} \]

Why is this called ‘quasi-affine’?
Presburger Sets: Reasoning

What do the resulting sets have to do with polyhedra? When are they convex?

Why polyhedra? Why not just rectangles?