Plan for Today

Look at the course web page

Types of Performance Optimizations

LULESH Intro

LLVM Intro

Concepts

Next Time

Types of Performance Optimizations

Definition

A performance optimization is a transformation that is expected to improve the program in some way; often consists of analysis and transformation e.g., decreasing the execution time or decreasing memory requirements

Machine-independent optimizations

– Eliminate redundant computation
– Move computation to less frequently executed place
– Specialize some general purpose code
– Remove useless code
– Identify parallelism and data reuse

Machine-dependent optimizations

– Replace costly operation with cheaper one
– Replace sequence of operations with cheaper one
– Hide latency
– Improve data locality
– Exploit machine parallelism
– Reduce power consumption

Enabling transformations

– Expose opportunities for other optimizations
– Help structure optimizations

Types of Optimizations (cont)

Sample Optimizations

Loop-invariant code motion

– e.g., for i = 1 to 10 do
  \[ x = 3; \]
  \[ \ldots \] 
  for i = 1 to 10 do

  Loop unrolling

– e.g., for i = 1 to 10 do 
  \[ a[i] = a[i] + 1; \]
  \[ a[i+1] = a[i+1] + 1; \]
  for i = 1 to 10 by 2 do
  \[ a[i] = a[i] + 1; \]
  \[ a[i+1] = a[i+1] + 1; \]
More examples: Loop Permutation for Improved Locality

Sample code: Assume Fortran’s Column Major Order array layout

```
do j = 1,6
   do i = 1,5
      A(j,i) = A(j,i) + 1
   enddo
endo

```

More examples: Parallelization

Can we parallelize the following loops?

```
do i = 1,100
   A(i) = A(i) + 1
endo
```

Yes

```
do i = 1,100
   A(i) = A(i-1) + 1
endo
```

No

Is an Parallelization or Optimization Worthwhile?

Criteria for evaluating optimizations

- Safety: does it preserve behavior?
- Profitability: does it actually improve the code?
- Opportunity: is it widely applicable?
- Cost (compilation time): can it be practically performed?
- Cost (complexity): can it be practically implemented?

How do we automate the process?

Represent programs in representations (e.g. LLVM)

- `a = b + c` // info about types of a, b, and c
- `d = load *a`
- Typed, 3-address code

Program analyses needed to find data dependences between computations

- Data-flow analyses (dependencies between scalar variables)
- Data-dependence analysis (dependencies between iterations in loops)

Specifying loop transformations and parallelization in the polyhedral framework

- Represent the loops within an integer tuple space
- Represent transformations as transformations in the integer tuple space
Representing Computations with Iteration Spaces

Idea

– Explicitly represent the iterations of a loop nest

Example

\[
\begin{align*}
do \ & i = 1, 6 \\
& \quad do \ j = 1, 5 \\
& \quad \quad A(i, j) = A(i-1, j-1) + 1 \\
& \quad enddo \\
& enddo
\end{align*}
\]

Iteration Space

– A set of integer tuples that represents the iterations of a loop
– Can visualize the dependences in an iteration space

Loops as Polyhedra, Transformations as Affine Functions

Original code

\[
\begin{align*}
do \ & i = 1, 6 \\
& \quad do \ j = 1, 5 \\
& \quad \quad A(i, j) = A(i-1, j+1) + 1 \\
& \quad enddo \\
& enddo
\end{align*}
\]

Loop Skewing Transformation

\[
\begin{bmatrix}
1 & 0 \\
1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
\end{bmatrix}
= 
\begin{bmatrix}
i+j \\
\\
\end{bmatrix}
\]

Bounds

\[
\begin{bmatrix}
-1 & 0 \\
0 & -1 \\
-1 & 0 \\
0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
\end{bmatrix}
\leq 
\begin{bmatrix}
-6 \\
-5 \\
-6 \\
-5 \\
\end{bmatrix}
\]

Translated code

\[
\begin{align*}
do \ & i' = 1, 6 \\
& \quad do \ j' = 1+i', 5+i' \\
& \quad \quad A(i', j'-i') = A(i'-1, j'-i'+1) + 1 \\
& \quad enddo \\
& enddo
\end{align*}
\]

Plan for Today

Look at the course web page

Types of Performance Optimizations

LULESH Intro

LLVM Intro

Concepts

Next Time

LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics

– Proxy application designed for DOE co-design projects
– Hydrodynamics consumes 27% of data center resources through DOD
– Solves a simple Sedov blast problem that has analytic answers
– Uses a regular cartesian mesh, but implemented using unstructured data structures

See PA1 writeup for how to compile and run it

– We will be learning more about this proxy application as the semester progresses.
Intermediate Representation and Compiler Infrastructure

- Typed, 3-address code
- Control-flow graphs with basic blocks
- Static Single Assignment (SSA)

Compiler Infrastructure: Tools, Analysis and Transformation Passes

- clang – frontend that translates C/C++ into LLVM bit code
- opt – analyze and transform LLVM bitcode
- llc – code generator from LLVM bitcode to native code

A Low-Level IR: 3-address code

We want to do analysis on an Intermediate Representation: 3-address code

- Linear representation: assignments, labels, (conditional) jumps
- Typically language-independent and nearly corresponds to machine instructions, difference: no stack code, but (temporary) variables
- There are named variables (parameters, variables) and temporaries (expressions). We can name the temporaries assuming unbounded number of (symbolic) registers.

Example operations

- (Indexed) copy \( x = y[i], y[i] = x, x = z, t1 = t2 \)
- Unary / binary op \( x = op z, x = v op z, t1 = t2 op t3 \)
- Address of \( p = & v \)
- Dereference \( x = * p, * p = x \)
- Pass param \( \text{param t0} \)
- Call \( t1 = \text{call f, l} \)
- (conditional) Branch \( \text{goto L1, if t1 goto L1} \)

LLVM Pass Infrastructure

Analyses and Transformations are Passes

Dependencies between passes

- When registered passes indicate which other passes they depend upon by overriding the `getAnalysisUsage` method
- The PassManager figures out the dependency Graph

Each Pass returns a boolean

- True indicates the pass has made a changed to the program.
- False indicates that no change was made.
- The PassManager runs until everyone stops.

Pass classes to derive from

- FunctionPass, used in PA1
- LoopPass, visits loops
- BasicBlockPass, visits basic blocks
- … some others

LLVM (a typed 3-address code)

Example operations

- (Indexed) copy \( x = y[i], y[i] = x, x = z, t1 = t2 \)
  - IN LLVM: \( %1 = %2, \text{how are array accesses done?} \)
- Unary / binary op \( x = op z, x = v op z, t1 = t2 op t3 \)
  - IN LLVM: \( \%\text{cmp} = \text{icmp sgt i32} %a, %b \)
  - Address of \( p = & v \)
  - IN LLVM: \( \text{does not appear to have this} \)
- Dereference \( x = * p, * p = x \)
  - IN LLVM: \( \%\text{tmp} = \text{load i32*} %X \)
- Pass param \( \text{param t0} \)
  - IN LLVM: \( \text{see function calls} \)
- Call \( t1 = \text{call f, l} \)
  - IN LLVM: \( \text{call void %foo(i8 97 signext)} \)
- (conditional) Branch \( \text{goto L1, if t1 goto L1} \)
  - IN LLVM: \( \text{br il %cmp, label %if.then, label %if.else} \)
**LLVM Program Organization**

**Program Representation Organization**
- A Module is a compilation unit that contains functions and global variables and constants.
- A Function represents a function definition and contains basic blocks
- Basic Blocks contain Instructions
- Instructions contain operands, each of which is a Value.
- All of the above are Values except for Modules

---

**Control Flow Graphs (CFGs)**

**Definition**
- A CFG is a graph whose nodes represent program statements and whose directed edges represent control flow

**Example**
1. \( a := 0 \)
2. \( b := a + 1 \)
3. \( c := c + b \)
4. \( a := b * 2 \)
5. \( \text{if } a < 9 \text{ goto L1} \)
6. \( \text{return } c \)

---

**Basic Blocks**

A Basic Block is a sequence of instructions with only one entry point and only one exit point.

**Example**
1. \( a := 0 \)
2. \( b := a + b \)
3. \( \text{L1: } c := b / d \)
   - \( c < x? \)
4. \( e := b / c \)
5. \( f := e + 1 \)
6. \( \text{L2: } g := f \)
7. \( h := t - g \)
8. \( \text{if } e > 0 \text{ goto L3} \)
9. \( \text{goto L1} \)
10. \( \text{L3: return} \)

---

**LLVM Program Organization**

**Program Representation Organization**
- A Module is a compilation unit that contains functions and global variables and constants.
- A Function represents a function definition and contains basic blocks
- Basic Blocks contain Instructions
- Instructions contain operands, each of which is a Value.
- All of the above are Values except for Modules

**Many LLVM passes use the SSA representation**
- Static single assignment
- Manipulating values instead of variable accesses
- The –mem2reg pass in opt creates the SSA representation.
- To understand SSA let’s first talk about data dependencies.
### Data Dependence

**Definition**
- Data dependences are constraints on the order in which statements may be executed.
- We say statement $s_2$ depends on $s_1$.
  - **Flow (true) dependence**: $s_1$ writes memory that $s_2$ later reads (RAW).
  - **Anti-dependence**: $s_1$ reads memory that $s_2$ later writes (WAR).
  - **Output dependences**: $s_1$ writes memory that $s_2$ later writes (WAW).
  - **Input dependences**: $s_1$ reads memory that $s_2$ later reads (RAR).

**True dependences**
- Flow dependences represent actual flow of data.

**False dependences**
- Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated.

### Representing Data Dependences

**Implicitly**
- Using variabledefs and uses.
  - Pros: simple.
  - Cons: hides data dependence (analyses must find this info).

**Def-use chains (du chains)**
- Link each def to its uses.
  - Pros: explicit; therefore fast.
  - Cons: must be computed and updated, space consuming.

**Alternate representations**
- *e.g.*, Static single assignment form (SSA), Program Dependence Graph (PDG), dependence flow graphs (DFG), value dependence graphs (VDG).

### DU Chains

**Definition**
- Du chains link each def to its uses.

**Example**

```
s1: a = b;
s2: b = c + d;
s3: e = a + d;
s4: b = 3;
s5: f = b * 2;
```
**UD Chains**

**Definition**
- ud chains link each use to its defs

**Example**

\[
\begin{align*}
s_1 & : a = b; \\
s_2 & : b = c + d; \\
s_3 & : e = a + d; \\
s_4 & : b = 3; \\
s_5 & : f = b * 2;
\end{align*}
\]

**Role of Alternate Program Representations**

**Advantage**
- Allow analyses and transformations to be simpler & more efficient/effective

**Disadvantage**
- May not be “executable” (requires extra translations to and from)
- May be expensive (in terms of time or space)

**Process**

Original Code (RTL) \[ \rightarrow \] Optimized Code (RTL)

\[
\begin{align*}
\text{Original Code} & \quad \rightarrow \quad \text{Optimized Code} \\
\text{SSA Code1} & \quad \rightarrow \quad \text{SSA Code2} \\
& \quad \rightarrow \quad \text{SSA Code3}
\end{align*}
\]

**Static Single Assignment (SSA) Form**

**Idea**
- Each variable has only one static definition
- Makes it easier to reason about values instead of variables
- Similar to the notion of functional programming

**Transformation to SSA**
- Rename each definition
- Rename all uses reached by that assignment

**Example**

\[
\begin{align*}
v := & \ldots \\
\ldots := & \ldots v \ldots \\
v := & \ldots \\
\ldots := & \ldots v \ldots
\end{align*}
\]

**What do we do when there’s control flow?**

**SSA and Control Flow**

**Problem**
- A use may be reached by several definitions

\[
\begin{align*}
v_0 := & \ldots \\
\ldots := & \ldots v_0 \ldots \\
v_1 := & \ldots \\
\ldots := & \ldots v_1 \ldots
\end{align*}
\]
SSA and Control Flow (cont)

Merging Definitions
– $\phi$-functions merge multiple reaching definitions

Example

```
 1
 v₀ := ...

 2
 v₁ := ...

 3
 4
 v₂ := $\phi$(v₀, v₁)
 ...v₂...
```

Another Example

```
 1
 v := 1

 2
 v := v+1

 3
 v₀ := 1

 4
 v₁ := $\phi$(v₀, v₂)
 v₂ := v₁+1
```

SSA vs. ud/du Chains

SSA form is more constrained

Advantages of SSA
– More compact
– Some analyses become simpler when each use has only one def
– Value merging is explicit
– Easier to update and manipulate?

Furthermore
– Eliminates false dependences (simplifying context)

```for (i=0; i<n; i++)
A[i] = i;
for (i=0; i<n; i++)
print(foo(i));
```

Unrelated uses of 1 are given different variable names

Concepts

Example Program Optimizations
– Loop-invariant code motion
– Induction variable elimination
– Loop unrolling, permutation, and parallelization

Approach for Program Analysis and Transformation
– Represent code in typed, 3-address code and perform data-flow analysis and transformation.
– Represent loops with iteration spaces and transform those.

LULESH: Regular versus Irregular Code

Intermediate Representations: LLVM
– Typed, 3-address code
– Basic blocks and control flow graphs: optimization scope
– Pass architecture used in compiler infrastructures and LLVM
– Data dependencies, Def-Use Chains, and SSA
Next Time

Reading
- The Architecture of Open Source Applications: LLVM, by Chris Lattner
- Look up new terms on the internet. Also ask questions on the CS553 Discussion Forum in RamCT.

Homework
- By Friday’s 3-4 lab session, work through Visualizing the Code in PA1
- LLVM reading assignment on the progress page

Friday Lab Session 3-4pm in 325 (here)
- Demonstrate a way to develop in LLVM.
- Come share the approach you are using!