PROGRAMMING AT A HIGH-LEVEL ON MULTI-CORES

What is a compiler to do?

Arun Chauhan
Indiana University
The Multi-core crisis

• Physical limitations
  ★ Transistor sizes
  ★ Clock skew

• Power consumption

• “Moore’s Law”
Software Productivity: The Real Crisis

- New software development
  - Programming models
  - Programming techniques
  - Programming languages

- Porting legacy code
  - Starting point: sequential or parallel?
  - Port optimized code
  - Source vs binary
Possible Solutions

• Novel languages
  ★ DARPA HPCS

• Extending traditional languages
  ★ Co-Array Fortran
  ★ UPC

• Libraries
  ★ ScalaPACK, MATAB*P

• High-level “scripting” languages
High-Level Scripting Languages

- Available and in use
- Modern
  - Support modern software engineering practices
- More powerful and general than libraries
- Programmers available
High-Level Scripting Languages

• Available and in use
• Modern
  ★ Support modern software engineering practices
• More powerful and general than libraries
• Programmers available

Can they solve the multi-core programming crisis?
Example: NASMG in MATLAB

\[ m = f(1) \cdot (n(c,c,c)) + \ldots \]
\[ f(2) \cdot (n(c,c,u)+n(c,c,d)+n(c,u,c)+n(c,d,c)+n(u,c,c)+n(d,c,c)) + \ldots \]
\[ f(3) \cdot (n(c,u,u)+n(c,u,d)+n(c,d,u)+n(c,d,d)+n(u,c,u)+n(u,c,d)+n(d,c,u)+n(d,c,d)+n(u,u,c)+n(u,d,c)+n(d,u,c)+n(d,d,c)) + \ldots \]
\[ f(4) \cdot (n(u,u,u)+n(u,u,d)+n(u,d,u)+n(u,d,d)+n(d,u,u)+n(d,u,d)+n(d,d,u)+n(d,d,d)); \]
Effects of Memory Optimizations on NAS MG

- Loop fusion
- Loop fusion + subscript opt.

Speedup over MATLAB interpreter

Core 2 Duo:
- Loop fusion: 3x
- Loop fusion + subscript opt.: 7x

Pentium IV:
- Loop fusion: -
- Loop fusion + subscript opt.: 6x
Why Compilation is Unnecessary

• Most of the computation takes place in libraries

• Interpretive overheads insignificant with byte-code

• Just-in-time compilation does a good job
Why Compilation is Unnecessary

• Most of the computation takes place in libraries
  ★ True for some applications, but not for many others
  ★ Parallelization on heterogeneous platforms

• Interpretive overheads insignificant with byte-code

• Just-in-time compilation does a good job
Why Compilation is Unnecessary

• Most of the computation takes place in libraries
  ★ True for some applications, but not for many others
  ★ Parallelization on heterogeneous platforms

• Interpretive overheads insignificant with byte-code
  ★ Byte-code does not eliminate library call overheads

• Just-in-time compilation does a good job
Why Compilation is Unnecessary

• Most of the computation takes place in libraries
  ★ True for some applications, but not for many others
  ★ Parallelization on heterogeneous platforms

• Interpretive overheads insignificant with byte-code
  ★ Byte-code does not eliminate library call overheads

• Just-in-time compilation does a good job
  ★ JIT compiler operates at byte-code level, missing many opportunities at high-level
Why Compilation is Necessary
Why Compilation is Necessary

Interprocedural Optimization
Benefits of Source-level Compilation

• Specialization
  ★ Type-based specialization can reduce or eliminate function call overheads

• Library function selection
  ★ Sequences of operations can be implemented efficiently

• Memory footprint reduction
  ★ Intermediate arrays and array computations can be eliminated

• Parallelization
  ★ Macro-operations provide naturally coarse granularity
Benefits of Source-level Compilation

- Specialization
  - Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - Intermediate arrays and array computations can be eliminated
- Parallelization
  - Macro-operations provide naturally coarse granularity
Benefits of Source-level Compilation

- Specialization
  - Type-based specialization can reduce or eliminate function call overheads

- Library function selection
  - Sequences of operations can be implemented efficiently

- Memory footprint reduction
  - Intermediate arrays and array computations can be eliminated

- Parallelization
  - Macro-operations provide naturally coarse granularity
Specialization: ARPACK Dense Matrix Kernel
Specialization: ARPACK Sparse Matrix Kernel

![Bar chart comparing running times of Matlab, ARPACK, and LibGen for sparse symmetric and nonsymmetric matrices.](image)
Type-based Specialization: DSP

jakes: Type–specialized FORTRAN vs MATLAB

- MATLAB 6.x
- MATLAB 5.3
- FORTRAN

<table>
<thead>
<tr>
<th>Platform</th>
<th>MATLAB 6.x</th>
<th>MATLAB 5.3</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun SPARC 336MHz</td>
<td>14.5</td>
<td>13.8</td>
<td>12.3</td>
</tr>
<tr>
<td>SGI Origin</td>
<td>9.0</td>
<td>9.2</td>
<td>8.5</td>
</tr>
<tr>
<td>Apple PowerBook G4 667MHz</td>
<td>11.0</td>
<td>10.8</td>
<td>9.0</td>
</tr>
</tbody>
</table>
Benefits of Source-level Compilation

• **Specialization**
  - Type-based specialization can reduce or eliminate function call overheads

• **Library function selection**
  - Sequences of operations can be implemented efficiently

• **Memory footprint reduction**
  - Intermediate arrays and array computations can be eliminated

• **Parallelization**
  - Macro-operations provide naturally coarse granularity
Benefits of Source-level Compilation

- Specialization
  - Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - Intermediate arrays and array computations can be eliminated
- Parallelization
  - Macro-operations provide naturally coarse granularity
Library Function Selection: Vector Outer-product

Implementing Vector Outer Product \((x^*y + A)\)

- AMD Opteron
- PowerPC 970 (Apple G5)
- Intel Xeon
- Intel Itanium 2

Vector size (thousands of double elements)

time(DGEMM) / time(DGER)
Implementing Scaled Vector Addition ($alpha \times x + beta \times y$)

- **AMD Opteron**
- **PowerPC 970 (Apple G5)**
- **Intel Xeon**
- **Intel Itanium 2**

**Vector size (millions of double elements)**

- **time(DAXPY+DSCAL) / time(DGEMM)**

**Library Function Selection: Scaled Vector Add**
Benefits of Source-level Compilation

• Specialization
  - Type-based specialization can reduce or eliminate function call overheads

• Library function selection
  - Sequences of operations can be implemented efficiently

• Memory footprint reduction
  - Intermediate arrays and array computations can be eliminated

• Parallelization
  - Macro-operations provide naturally coarse granularity
Benefits of Source-level Compilation

- Specialization
  - Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - Intermediate arrays and array computations can be eliminated
- Parallelization
  - Macro-operations provide naturally coarse granularity
Temporary Arrays: Matrix Expressions

\[ A + A \times B' + 2 \times (A + B)' \times A + (x + y) \times x' \]

OR

\[ A + A \times B' + 2 \times A' \times A + 2 \times B' \times A + x \times x' + y \times x' \]
Parenthesized vs Distributed

Implementing A Big Expression

- AMD Opteron
- PowerPC 970 (Apple G5)
- Intel Xeon
- Intel Itanium 2

Vector size (thousands of double elements)

time(parenthesized) / time(distributed)
Absolute Time Differences on Itanium 2

-600  -400  -200  0  200  400  600  800  1000  1200  1400

Vector size (thousands of double elements)
Temporary Arrays: Matrix Expressions

\[ A + A \cdot B' + 2 \cdot (A + B)' \cdot A + (x + y) \cdot x' \]

OR

\[ A + A \cdot B' + 2 \cdot A' \cdot A + 2 \cdot B' \cdot A + x \cdot x' + y \cdot x' \]
Temporary Arrays: Matrix Expressions

\[ A + A \cdot B' + 2 \cdot (A+B)' \cdot A + (x+y) \cdot x' \]

\[
\begin{align*}
&\text{copy}(A,tmp0); \\
&\text{gemm}(1,A,B,1,tmp0); \\
&\text{copy}(A,tmp1); \\
&\text{axpy}(1,B,1,tmp1); \\
&\text{gemm}(2,tmp1,A,1,tmp0); \\
&\text{copy}(x,tmp1); \\
&\text{axpy}(1,y,1,tmp1); \\
&\text{ger}(1,tmp1,x,tmp0);
\end{align*}
\]

\[ A + A \cdot B' + 2 \cdot A' \cdot A + 2 \cdot B' \cdot A + x \cdot x' + y \cdot x' \]

\[
\begin{align*}
&\text{copy}(A,tmp0); \\
&\text{gemm}(1,A,B,1,tmp0); \\
&\text{copy}(A,tmp1); \\
&\text{axpy}(1,B,1,tmp1); \\
&\text{gemm}(2,tmp1,A,1,tmp0);
\end{align*}
\]
Function Selection Algorithm

```
algorithm basic-block-function-selector
inputs: P = Octave source code (as AST)
        S = SSA graph of P
        L = target library
outputs: R = Modified version of P with operations
         mapped to the function calls in L
         whenever possible

set R to an empty AST
for each simple statement, s, in P, do
  if (s does not have an operation implemented by L)
    add s unchanged to R
  else
    let ⊗ be the operation in s
    let ω be the optimal choice of function in L
    implementing ⊗ based on the current context
    if (ω is a multi-op function and
        ⊗ is a candidate operation)
      for each operand u
        let d be the statement defining u, obtained from S
        if (d can be subsumed in ω)
          add the operands of d to ω
      endif
    endif
  endif
endfor
endif
add the call to ω to R
endif
endfor
```
MEMORY BEHAVIOR MODELING
Motivation

The traditional theoretical approach to analysis involves counting basic operations performed on an abstract computer. These operations are generic abstractions of CPU instructions, each assumed to have some unit cost. However, on modern architectures, instructions have varying costs; in particular, a memory access can be several orders of magnitude more time consuming than any single machine instruction, depending on where the operands are located. New abstract models are needed to describe these new types of costs.

Catherine C. McGeoch,
Experimental Algorithmics,
Communications of the ACM Volume 50, Number 11 (2007), Pages 27-31
Reuse Distance (courtesy: Chen Ding @ Rochester)

- **Reuse distance** of an access to data \( d \)
  - the volume of data between this and the previous access to \( d \)
- **Reuse signature** of an execution
  - the distribution of all reuse distances
  - gives the miss rate of fully associative cache of all sizes

![Graph showing reuse distances and references]

Mattson, Gecsei, Slutz, Traiger
Source-level Reuse Distance

The source-level reuse distance between two memory references (to the same location) is the volume of the source-level data accessed between the two memory references.

\[
x = a + b; \\
c = a + d[i] \times 100; \\
y = x;
\]

Distance = 6 = Five variables \((a, b, c, d, i)\) + one constant \((100)\)
Complex Expressions

\[
C = x + \text{foo}(i,j) \times B[i+j,10]; \\
D = \text{bar}(i,j) + x + y;
\]

\[
t_1 = i + j; \\
t_2 = \text{foo}(i,j); \\
t_3 = t_2 \times B[t_1, 10]; \\
C = x + t_3; \\
t_4 = \text{bar}(i, j); \\
t_5 = t_4 + x; \\
D = t_5 + y;
\]
Reuse from Dependence Information

- Dependence for code transformations
  - Between two references to the same memory location
  - One of the references is a write

- Dependence for reuse: drop the write requirement
  - True dependence ($\delta$)
  - Anti-dependence ($\delta^{-1}$)
  - Output dependence ($\delta^o$)
  - Input dependence ($\delta^i$)
Reuse within a Simple Statement

\[ [x, u, x] = s_{\text{func}}(u, y, w, w) \]

\[ \text{dist} = 0 \]
## Computing Reuse within a Basic Block

**Algorithm Compute_R_simple**

INPUT: basic block $B$, dependence graph $D$ restricted to $B$

OUTPUT: $R_{p_1, p_2} \forall$ pairs of ref. points $p_1$ and $p_2$ in $B$ that are successive accesses of the same memory location

BEGIN

1. let $N_v$ be the unique number associated with the vertex $v$ in $D$ induced naturally by the total ordering of ref. points in $B$
2. let the length of an edge, $e = v_1 \rightarrow v_2$ be defined as $N_{v_2} - N_{v_1}$

3. **foreach** edge $e = v_1 \rightarrow v_2$ in $D$ sorted by length
4. **if** (either $v_1$ or $v_2$ is unexamined)
5. mark $v_1$ and $v_2$ as “examined”
6. $R_{v_1,v_2} = N_{v_2} - N_{v_1} -$ number of edges lying wholly between $v_1$ and $v_2$

END
Reuse with Forward Control Flow through the reference points

Suppose that there is a dependence edge that starts at a point in the block and goes to a reference point or end at the same block. If there is a dependence edge that starts at a point and goes to a reference point in another block, then there must be dependence edges connecting to the earliest reference point of all that are successive accesses of the same memory location in the block. Thus, in Figure 5, the dependence graph is now restricted to those edges that run across basic blocks and that either start or end at "globals" in the sense of Cooper and Torczon's algorithm for building the semi-pruned Static Single Assignment (SSA) form of a program.

In extending the simple algorithm to DAGs we will assume that the DAG always has a unique start node. If the DAG forms a Directed Acyclic Graph (DAG), we will extend the algorithm to code regions larger than basic blocks. Specifically, we allow forward control flow without any back edges (loops). Such a control flow graph with only forward control flow edges is an example of a piece of code in the form of a control flow graph with two-way or multi-way branches.

Figure 5 shows a control flow graph with only forward control flow edges. Edges in the figure represent basic blocks that have been pruned to leave only the edge assignment (SSA) form of a program. The dependence graph is now restricted to those edges that run across basic blocks and that either start or end at "globals". The remaining blue edges will be ignored. The remaining edges are similarly pruned to leave only the edge that corresponds to the maximum reuse distance along the different paths.

For example, edges in the increasing order of their lengths do not guarantee the closest ones. For example, edges in the increasing order of their lengths do not guarantee the closest ones. For example, edges in the increasing order of their lengths do not guarantee the closest ones. For example, edges in the increasing order of their lengths do not guarantee the closest ones.

As before, a subset of dependence edges provide reuse distances between such points. The reuse distances along the different paths should be simply ignored the control flow and add the unique start node can always be added.

However, with additional information, coming from profiling or user input, the reuse distances along the different paths should simply be ignored the control flow and add the unique start node can always be added. Rectangles in the figure represent basic blocks that have been pruned to leave only the edge that corresponds to the maximum reuse distance along the different paths.

Incomparable pairs of vertices in the figure will be ignored. The remaining edges are similarly pruned to leave only the edge that corresponds to the maximum reuse distance along the different paths. However, simply examining the dependence edges in the dependence graph forms a Directed Acyclic Graph (DAG). We will extend the algorithm to code regions larger than basic blocks. Specifically, we allow forward control flow without any back edges (loops). Such a control flow graph with only forward control flow edges is an example of a piece of code in the form of a control flow graph with two-way or multi-way branches.

Figure 5 shows a control flow graph with only forward control flow edges. Edges in the figure represent basic blocks that have been pruned to leave only the edge assignment (SSA) form of a program.
Computing Reuse with Forward Control Flow

**Algorithm Compute_R_With_Branches**

**Input:** control flow graph $G$ with weighted edges, dependence graph $D$

**Output:** $R_{p_1,p_2}$ for all pairs of ref. points $p_1$ and $p_2$ in $B$ that are successive accesses of the same memory location

**begin**
1. restrict $D$ to cross-block edges such that only the first incoming and the last outgoing edge is retained for each sequence of dependence edges
2. **foreach** block, $b$, in $G$
3. $(U[b],D[b]) = Exposed_Data_Volumes(b,D)$

**end**
4. **foreach** connected component, $C$, of $D$
5. **foreach** pair of vertices $v_1$ and $v_2$ in $C$
6. $P =$ set of control-flow nodes that lie on any path between $v_1$ and $v_2$ that does not go through any control-flow node containing any other vertex in $C$
7. if $(P$ is not empty)$\quad R_{v_1,v_2} = Weighted_Dist(v_1,v_2,P,U,D,G)$

**end**
**end**
**end**
Reuse with Reverse Control Flow (Loops)

These dependence edges reduce the unique memory access count

$P_i$  $Q_j$

$\ldots$  $\ldots$
Challenges

• Correlation between source-level and binary reuse distances
• Efficient estimation of source-level reuse distances
• Composition
  ★ Bottom up computation
  ★ Empirical methods for “black box” libraries
• Effective and efficient summarization
• Code optimization using the source-level reuse distance information
RUBY
Ruby on One Slide

• Fully object oriented
  ★ Derived from Smalltalk
  ★ Everything is an object, including classes
  ★ No standard operators

• Powerful meta-programming support
  ★ Classes and objects may be redefined
  ★ Methods may be (re/un)defined

• Advanced language features
  ★ Co-routines
  ★ Continuations
  ★ Blocks (generalized anonymous functions)
Ruby is Slow!

 Courtesy: *Zen and the Art of Programming*
 By Antonio Cangiano, Software Engineer & Technical Evangelist at IBM
Current Efforts on Ruby

- Ruby 1.8: The current stable implementation, also known as the MRI (Matz Ruby Interpreter)
- Ruby 1.9: The YARV based implementation of Ruby, faster with language improvements, but incompatible with 1.8
- JRuby: Ruby on the Java VM, both interpreter and compiler to Java bytecode (Ruby 1.8)
- XRuby: Ruby to Java bytecode compiler (Ruby 1.8)
- IronRuby: Ruby on the Microsoft CLR (Ruby 1.8)
- Rubinius: Ruby compiler based on the Smalltalk 80 compiler (Ruby 1.8)
- MacRuby: MacRuby, port of Ruby 1.9 to Objective-C, using Objective-C objects to implement Ruby and the Objective-C 2.0 garbage collector (Ruby 1.9)
Partially Evaluating Ruby

- Bottom-up approach
  - Tabulate the “safe” primitives
    - Most are written in C
  - Partially evaluate include libraries
  - Partially evaluate the user code

- Target applications
  - Ruby on Rails
  - Home-grown graduate student database management

- Does not work for libraries
- Too much redundant effort
  - Can we partially evaluate libraries conditionally?
Current Status

• Ruby front-end
• C front-end for semi-automatically classifying primitives as “safe” for partial evaluation
• Software infrastructure for partial evaluation by interfacing with Ruby interpreter
Other Projects

• Declarative approach to parallel programming
  ★ Let users specify parallelism
  ★ Separate computation from communication
  ★ Funded by NSF two days ago!

• MPI-aware compilation
  ★ Optimizing legacy MPI code for multicore clusters

• Distributed VM for Ruby

• Parallelizing for heterogeneous targets
THANK YOU!

http://www.cs.indiana.edu/~achauhan/
http://phi.cs.indiana.edu/