Overview

- Why MATLAB?
- Overview of the McLab tools
- McVM — a Virtual Machine and Just-In-Time (JIT) compiler
- McFOR — translating MATLAB to FORTRAN95

Culture Gap

Scientists / Engineers
- Comfortable with informal descriptions and "how to" documentation.
- Don’t really care about types and scoping mechanisms, at least when developing small prototypes.
- Appreciate libraries, convenient syntax, simple tool support, and interactive development tools.

Programming Language / Compiler Researchers
- Prefer more formal language specifications.
- Prefer well-defined types (even if dynamic) and well-defined scoping and modularization mechanisms.
- Appreciate “harder/deeper/more beautiful” programming language/compiler research problems.

Goals of the McLab Project

- Improve the understanding and documentation of the semantics of MATLAB.
- Provide front-end compiler tools suitable for MATLAB and language extensions of MATLAB.
- Provide a flow-analysis framework and a suite of analyses suitable for a wide range of compiler/soft. eng. applications.
- Provide back-ends that enable experimentation with JIT and ahead-of-time compilation.

Enable PL, Compiler and SE Researchers to work on MATLAB
McLAB – Overall Structure

McLAB Extensible Front-end

Analysis Engine

How to build the back-ends?

Back-end #1: MATLAB generator

- McLAB can be used as a source-to-source translator:
  - source-level optimizations like loop unrolling
  - source-level refactoring tool
- McLAB can generate:
  - xml files (used to communicate with McVM)
  - text .m files
- Comments are maintained to enable readable output.

Back-end #2: McVM with JIT compiler

- Currently being extended by Nurudeen Lameed and Rahul Garg, Ph.D. candidates, McGill.
McVM-McJIT

- The dynamic nature of MATLAB makes it very suitable for a VM/JIT.
- MathWorks' implementation does have a JIT, although technical details are not known.
- McVM/McJIT is an open implementation aimed at supporting research into dynamic optimization techniques for MATLAB.

Design Choices for JITs

1. Interpreter + JIT compiler with various levels of optimizations.
2. Fast JIT for naïve code generation + optimizing JIT with various levels of optimizations.

- McVM uses the 1st option because it simplifies adding new features, if a feature is not yet supported by the JIT it can back-up to the interpreter implementation, which is easy to provide.

McVM Design

- A basic, but fast, interpreter for the MATLAB language.
- A garbage-collected JIT Compiler as an extension to the interpreter.
- Easy to add new data types and statements by modifying only the interpreter.
- Supported by the LLVM compiler framework and some numerical computing libraries.
- Written entirely in C++; interface with the McLAB front-end via a network port.

McVM Organization

- Designed with a modular architecture that allows for easy integration of new features and optimizations.
- Utilizes the LLVM compiler framework for efficient code generation.
- Provides a clean interface for communication with McLAB front-end.
MATLAB Optimization Challenges

float sumvals(float start, float step, float stop)
{
    float i = start;
    float s = i;
    while (i < stop)
    {
        i = i + step;
        s = s + i;
    }
    return s;
}

function sumvals(start, step, stop)
    i = start;
    s = i;
    while i < stop
        i = i + step;
        s = s + i;
    end
end

function caller()
    a = sumvals(1, 1, 10^6);
    b = sumvals([1 2], [1.5 3], [20^5 20^5]);
    c = [a b];
    disp(c);
end

McJIT: Executing a Function

Just-In-Time Specialization (1)

Just-In-Time Specialization (2)

Just-In-Time Specialization (3)
Just-In-Time Specialization (4)

```matlab
function s = sumvals(start, step, n)
    a = start; % Initialize
    while a < n % Loop until a is greater than or equal to n
        s = s + a; % Add a to the sum
        a = a + step; % Increment a
    end
end
```

Just-In-Time Specialization - 2nd example

```matlab
>> a = sumvals(1, 1, 10^6);
>> b = sumvals([1 2], [1.5 3], [20^5 20^5]);
>> a = sumvals(1, 1, 500);
>> c = [a b];
>> disp(c);
```

JIT – second specialization (1)

```matlab
function s = sumvals(start <1x2 int>, step <1x2 int>, n <1x2 int>)
    a = start; % Initialize
    while a < n % Loop until a is greater than or equal to n
        s = s + a; % Add a to the sum
        a = a + step; % Increment a
    end
end
```

JIT – second specialization (2)

```matlab
function s = sumvals(start <1x2 int>, step <1x2 int>, n <1x2 int>)
    a = start; % Initialize
    while a < n % Loop until a is greater than or equal to n
        s = s + a; % Add a to the sum
        a = a + step; % Increment a
    end
end
```

JIT – third specialization same as first

```matlab
>> a = sumvals(1, 1, 10^6);
>> b = sumvals([1 2], [1.5 3], [20^5 20^5]);
>> a = sumvals(1, 1, 500);
>> c = [a b];
>> disp(c);
```

Type and Shape Inference

- In MATLAB, work with incomplete information
  - Dynamic loading: working with incomplete program
  - Dynamic typing: variables can change type
- Know argument types, what can we infer?
  - Propagate type info to deduce locals type and return type
- Forward dataflow analysis
  - Based on abstract interpretation
  - Structure-based fixed point
  - Annotates AST with type info
Flow Analysis Summary

- Start from known argument types
- Propagate type information forward
- Use a transfer function for each expression
  - Transfer functions provided for all primitive operators
  - Library functions provide their own transfer functions
  - Function calls resolved, recursively inferred
- Assignment statements can change var. types
- Merge operator
  - Union + filtering

Lattice of McVM types

- Top (Unknown type, could be any)
- Matrix like types
- Cell Array
- Matrix types
- Char array
- Logical array
- Double matrix
- Complex Matrix
- Bottom (No information inferred)

Type Abstraction Properties

- Collection of simple abstractions
- Specific features computed in parallel
- Represent variable types with 8-tuples:

```
<overallType, is2D, isScalar, isInteger,
sizeKnown, size, handle, cellTypes>
```

```
 type(a) =
    <overallType = double, is2D = T, isScalar = F,
isInteger = T, sizeKnown = T, size = (1,2),
handle = null, cellTypes = {}>,
    <overallType = double, is2D = T, isScalar = T,
isInteger = F, sizeKnown = T, size = (1,1),
handle = null, cellTypes = {}>,
    <overallType = char, is2D = T, isScalar = T,
isInteger = T, sizeKnown = T, size = (1,1),
handle = null, cellTypes = {}>
```
Transfer Functions

\[
a = 5 \times 10^{11};
\]
\[
b = 1.0e12 \times [0.8533 \ 1.7067];
\]
\[
c = [a \ b];
\]

\[
\text{type}(a) = \{
\begin{cases}
\text{overallType} = \text{double}, \\
\text{is2D} = \text{T}, \\
\text{isScalar} = \text{T}, \\
\text{isInteger} = \text{T}, \\
\text{sizeKnown} = \text{T}, \\
\text{size} = (1,1), \\
\text{handle} = \text{null}, \\
\text{cellTypes} = \{
\end{cases}
\}
\]

\[
\text{type}(b) = \{
\begin{cases}
\text{overallType} = \text{double}, \\
\text{is2D} = \text{T}, \\
\text{isScalar} = \text{F}, \\
\text{isInteger} = \text{F}, \\
\text{sizeKnown} = \text{T}, \\
\text{size} = (1,2), \\
\text{handle} = \text{null}, \\
\text{cellTypes} = \{
\end{cases}
\}
\]

\[
\text{type}([a \ b]) = \{
\begin{cases}
\text{overallType} = \text{double}, \\
\text{is2D} = \text{T}, \\
\text{isScalar} = \text{F}, \\
\text{isInteger} = \text{F}, \\
\text{sizeKnown} = \text{T}, \\
\text{size} = (1,3), \\
\text{handle} = \text{null}, \\
\text{cellTypes} = \{
\end{cases}
\}
\]

Experimental Results

- 20 benchmark programs
- FALCON, OTTER, etc. Some made by McLAB
- Measured
  - Dynamic availability of type info.
  - Number of versions compiled
  - Compilation time
    - 0.55s per benchmark, on average

Results of Type Analysis

Experimental setup
- Core 2 Quad Q6600, 4GB RAM
- Ubuntu 9.10, kernel 2.6.31, 32-bit
- All timings averaged over 10 runs
Comparing
- McVM interpreter, McVM JIT w/ spec.
- MATLAB R2009a
- GNU Octave 3.0.5
- McFor / GNU Fortran 4.4.1

Results (Speed-up vs Interpreters)
Results (Speed-up vs MATLAB, McFOR)

Results (Speed Results (Speed—up vs MATLAB, McFOR))

<table>
<thead>
<tr>
<th>Function</th>
<th>JIT vs MATLAB, Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>Vs MATLAB</td>
</tr>
<tr>
<td>nb3d</td>
<td>Vs Fortran</td>
</tr>
<tr>
<td>edit</td>
<td></td>
</tr>
<tr>
<td>adpt</td>
<td></td>
</tr>
<tr>
<td>diff</td>
<td></td>
</tr>
<tr>
<td>fdtd</td>
<td></td>
</tr>
<tr>
<td>mbrt</td>
<td></td>
</tr>
<tr>
<td>clos</td>
<td></td>
</tr>
<tr>
<td>crni</td>
<td></td>
</tr>
</tbody>
</table>
| benchmark takes 1321s to execute in McVM, 6.95s in MATLAB
- ~4x faster than Octave, but still slow
- Why?
  - Scalars known 68.7%, one of the lowest ratios
  - Unknown types propagated through entire benchmark
  - Weakness of type inference system to be fixed in future work

About Slow Benchmarks

Back-end #3: McFOR FORTRAN95 generator

- First version of McFOR based on Jun Li’s M.Sc. thesis, McGill.
- Current version under development by Anton Dubrau, M.Sc. candidate, McGill.

Goals of McFOR

- Handle as large a sub-set of MATLAB as possible, while staying in the "static" setting.
- Generate code that can be effectively compiled by modern FORTRAN compilers.
- Make the generated code readable by programmers.
- Allow longer compile times and whole program analysis.
- Limit the need for type annotations.

Challenges

- Determining which identifiers are variables and which are functions.
- Finding static types which match those of FORTRAN.
- Mapping high-level MATLAB array operations to the FORTRAN95 equivalents.
- Handling reshaping implicit in MATLAB operations, including concatenation.

Handling Incompatible Types

Alpha Nodes

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0;</td>
</tr>
<tr>
<td>if (i&gt;0)</td>
</tr>
<tr>
<td>S1: x = foo(i);</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>S2: x = bar(i);</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>x = alpha(S1,52);</td>
</tr>
<tr>
<td>y = x;</td>
</tr>
</tbody>
</table>

Eliminating Alpha Nodes

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0;</td>
</tr>
<tr>
<td>if (i&gt;0)</td>
</tr>
<tr>
<td>x = foo(i);</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>y = x;</td>
</tr>
<tr>
<td>x1 = bar(i)</td>
</tr>
<tr>
<td>y = x1;</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Value Range Propagation

MATLAB
\[
\begin{align*}
n &= \text{floor}(11.5); \quad // n = [n, n] \\
\text{for } i = 1:n \\
x &= 1+2*i; \quad // x = [3,1+2*n] \\
A(x) &= i; \\
\text{end}
\end{align*}
\]

FORTRAN
\[
\begin{align*}
n &= \text{floor}(10.5); \\
\text{DO } i = 1,n \\
x &= 1+(2*i); \\
\text{IF}(\text{.NOT.ALLOCATED(A)}) \text{ THEN} \\
\text{ALLOCATE(A((1+(2*n))));} \\
\text{END IF} \\
A(x) &= i; \\
\text{END DO} \\
n &= \text{fix}(n/2); \quad // n = [n, n] \\
A(n+1) &= n;
\end{align*}
\]

Related Work

- Procedure cloning: Cooper et al. (1992)
- MATLAB type inference: Joisha & Banerjee (2001)
  - Suggested for error detection
  - Source-to-source transformation
- MaJIC: JIT compilation and offline code cache (2002)
  - Speculative compilation MATLAB to C/Fortran
- Psyco: Python VM with specialization by need (2004)
  - TraceMonkey: JIT optimization of code traces (2009)

Ongoing Work

- McVM:
  - profile-guided optimization and re-optimization with on-stack replacement
  - target GPU/multi-core
- McFOR:
  - "decompile" to more programmer-friendly FORTRAN95
  - refactoring toolkit to help restructure "dynamic" features to "static" features

Conclusions

- McLAB is a toolkit to enable PL, Compiler and SE research for MATLAB
- front-end for language extensions
- analysis framework
- three back-ends including McVM and McFOR

http://www.sable.mcgill.ca/mclab

EXTRA SLIDES
Internal Intermediate Representation

- A simplified form of the Abstract Syntax Tree (AST) of the original source program
- It is machine independent
- All IIR nodes are garbage collected

IIR: A Simple MATLAB Program

```matlab
function [a] = test(n)
a = zeros(1, n);
for i = 1:n
    a(i) = i^2;
end
end
```

McVM Project Class Hierarchy (C++ Classes)

Supported Types

- Logical Arrays
- Character Arrays
- Double-precision floating points
- Double-precision complex number matrices
- Cell arrays
- Function Handles