Telescoping MATLAB for DSP Applications

PhD Thesis Defense

Arun Chauhan

Computer Science, Rice University
Two True Stories
Two True Stories

- the world of Digital Signal Processing
Two True Stories

• the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C

Two True Stories

- the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C

- linear algebra through MATLAB
Two True Stories

- the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C

- linear algebra through MATLAB
  - ARPACK—a linear algebra package to solve eigenvalue problems
  - prototyped in MATLAB
  - painfully hand translated to FORTRAN
Lessons
Lessons

• programming is an unavoidable fact of life to conduct research in science and engineering
Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages
Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages
- users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica
Lessons

• programming is an unavoidable fact of life to conduct research in science and engineering

• users do not like programming in traditional languages

• users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica

• performance problems limit their use
Lessons

• programming is an unavoidable fact of life to conduct research in science and engineering

• users do not like programming in traditional languages

• users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica

• performance problems limit their use

• the productivity connection
“It was our belief that if FORTRAN, during its first months, were to translate any reasonable ‘scientific’ source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.”

–John Backus
Pushing the Level Again
Pushing the Level Again

effective compilation
Pushing the Level Again

effective compilation

efficient compilation
It is possible to efficiently compile numerical programs written in high-level languages to achieve performance close to that achievable in a lower-level language.
Fundamental Observation

- libraries are the key in optimizing high-level scripting languages
  
  \[
  \text{a} = \text{x} \ast \text{y} \implies \text{a} = \text{MATMULT(x, y)}
  \]
Fundamental Observation

- libraries are the key in optimizing high-level scripting languages
  
  \[ a = x \times y \implies a = \text{MATMULT}(x, y) \]

- libraries **define** high-level languages!
  - a large effort in HPC is towards writing libraries
  - domain-specific libraries make scripting languages useful and popular
  - high-level operations are largely “syntactic sugar”
Libraries as Black Boxes
Libraries as Black Boxes

Whole Program Compilation

user program

library one
one.one
one.two

library two

compiler

object code
Telescoping Languages Approach

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts
Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts
- link appropriate versions into the user script
Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts
- link appropriate versions into the user script

analogous to offline indexing by search engines
Telescoping Languages: Entities

library writer

library compiler

end user

script compiler

Telescoping Languages: Entities

- **Library Writer**
  - Write libraries
  - Write annotations
  - Subroutine `VMP (C, Z, ..., s)`
  - "expect s to be mostly 1"

- **Library Compiler**

- **End User**

- **Script Compiler**

---

*PhD Thesis Defense: Telescoping MATLAB for DSP Applications*  
*July 10, 2003*
Telescoping Languages: Entities

- **library writer**
  - write libraries
  - write annotations
  - subroutine VMP (C, Z, ..., s)
    - "expect s to be mostly 1"

- **library compiler**
  - specialize code
  - VMP_step1 (C, Z, ...)

- **end user**

- **script compiler**

*PhD Thesis Defense: Telescoping MATLAB for DSP Applications*
Telescoping Languages: Entities

library writer

write libraries

subroutine VMP (C, Z, ..., s)

write annotations

“expect s to be mostly 1”

library compiler

specialize code

VMP_step1 (C, Z, ...)

end user

write script

call VMP (C, Z, ..., 1)
Telescoping Languages: Entities

library writer
- write libraries
- write annotations
- subroutine VMP (C, Z, ..., s)
  
  "expect s to be mostly 1"

library compiler
- specialize code
- VMP_step1 (C, Z, ...)

end user
- write script
- call VMP (C, Z, ..., 1)

script compiler
- choose optimized variant
- call VMP_step1 (C, Z, ...)

Telescoping Languages Approach

- Domain library
- Language building compiler
- Script
- Script translator
- Enhanced language compiler
- Optimized object program
Challenges

identifyingspecializationopportunities whichkinds of specializations howmany

identifyinghighpay-®optimizations mustbeapplicableintelescopinglanguagescontext shouldfocuson these¯rst

enablingthelibrarywritertoexpresssthese transformations guidethespecialization
Challenges

- identifying specialization opportunities
  - which kinds of specializations
  - how many
Challenges

- identifying specialization opportunities
  - which kinds of specializations
  - how many

- identifying high pay-off optimizations
  - must be applicable in telescoping languages context
  - should focus on these first
Challenges

• identifying specialization opportunities
  - which kinds of specializations
  - how many

• identifying high pay-off optimizations
  - must be applicable in telescoping languages context
  - should focus on these first

• enabling the library writer to express these transformations
  - guide the specialization
Developing the Compiler

- compile MATLAB
- emit specialized output code
- implement identified high-payoff optimizations
- implement newly discovered optimizations
Developing the Compiler

- compile MATLAB
- emit specialized output code
- implement identified high-payoff optimizations
- implement newly discovered optimizations

Example Compilation

function mcc_demo
    x = 1;
    y = x / 10;
    z = x * 20;
    r = y + z;
Example Compilation

```c
static void Mmcc_demo (void) {
    ...
    mxArray * r = NULL;
    mxArray * z = NULL;
    mxArray * y = NULL;
    mxArray * x = NULL;
    mlfAssign(&x, _mxarray0_); /* x = 1; */
    mlfAssign(&y, mclMrddivide(mclVv(x, "x"), _mxarray1_)); /* y = x / 10; */
    mlfAssign(&z, mclMtimes(mclVv(x, "x"), _mxarray2_)); /* z = x * 20; */
    mlfAssign(&r, mclPlus(mclVv(y, "y"), mclVv(z, "z"))); /* r = y + z; */
    mxDestroyArray(x);
    mxDestroyArray(y);
    mxDestroyArray(z);
    mxDestroyArray(r);
    ...
}
```
static void Mmcc_demo (void) {
    ...
    double r;
    double z;
    double y;
    double z;
    mlfAssign(&x, _mxarray0_); /* x = 1; */
    mlfAssign(&y, mclMrdivide(mclVv(x, ”x”), _mxarray1_)); /* y = x / 10; */
    mlfAssign(&z, mclMtimes(mclVv(x, ”x”), _mxarray2_)); /* z = x * 20; */
    mlfAssign(&r, mclPlus(mclVv(y, ”y”), mclVv(z, ”z”))); /* r = y + z; */
    mxDestroyArray(x);
    mxDestroyArray(y);
    mxDestroyArray(z);
    mxDestroyArray(r);
    ...
}
static void Mmcc_demo (void) {
    ...
    double r;
    double z;
    double y;
    double z;
    scalarAssign(&x, 1); /* x = 1; */
    scalarAssign(&y, scalarDivide(x, 10)); /* y = x / 10; */
    scalarAssign(&z, scalarTimes(x, 20)); /* z = x * 20; */
    scalarAssign(&r, scalarPlus(y, z)); /* r = y + z; */
    mxDestroyArray(x);
    mxDestroyArray(y);
    mxDestroyArray(z);
    mxDestroyArray(r);
    ...
}
static void Mmcc_demo (void) {
    ...
    double r;
    double z;
    double y;
    double z;
    x = 1; /* x = 1; */
    y = x / 10; /* y = x / 10; */
    z = x * 20; /* z = x * 20; */
    r = y + z; /* r = y + z; */
    /* mxDestroyArray(x); */
    /* mxDestroyArray(y); */
    /* mxDestroyArray(z); */
    /* mxDestroyArray(r); */
    ...
}
Inferring Types
(Joint work with Cheryl McCosh)
Inferring Types
(Joint work with Cheryl McCosh)

- \text{type} \equiv <\tau, \delta, \sigma, \psi>
  - \tau = \text{intrinsic type, e.g., int, real, complex, etc.}
  - \delta = \text{array dimensionality, 0 for scalars}
  - \sigma = \delta\text{-tuple of positive integers}
  - \psi = \text{“structure” of an array}
Inferring Types
(Joint work with Cheryl McCosh)

- type $\equiv <\tau, \delta, \sigma, \psi>$
  - $\tau =$ intrinsic type, e.g., int, real, complex, etc.
  - $\delta =$ array dimensionality, 0 for scalars
  - $\sigma =$ $\delta$-tuple of positive integers
  - $\psi =$ “structure” of an array

- type inference in general
  - type = “smallest” set of values that preserves meaning
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

- dimensionality constraints

\[
\begin{align*}
x &= 1 \\
y &= x / 10 \\
z &= x \times 20 \\
r &= y + z
\end{align*}
\]
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

- **dimensionality constraints**

\[
x = 1 \quad \text{LHS dims} = \text{RHS dims}
\]

\[
y = x / 10
\]

\[
z = x * 20
\]

\[
r = y + z
\]
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

• dimensionality constraints

\[ x = 1 \quad \text{LHS dims} = \text{RHS dims} \]

\[ y = x / 10 \quad (x, y \text{ scalar}) \text{ OR } (x, y \text{ arrays of same size}) \]

\[ z = x \ast 20 \]

\[ r = y + z \]
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

- dimensionality constraints

\[
x = 1 \quad \text{LHS dims = RHS dims}
\]

\[
y = \frac{x}{10} \quad (x, y \text{ scalar}) \text{ OR } (x, y \text{ arrays of same size})
\]

\[
z = x \times 20 \quad (x, z \text{ scalar}) \text{ OR } (x, z \text{ arrays of same size})
\]

\[
r = y + z
\]
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

- dimensionality constraints

\[ x = 1 \]
LHS dims = RHS dims

\[ y = x / 10 \]  
(x, y scalar) OR (x, y arrays of same size)

\[ z = x \times 20 \]  
(x, z scalar) OR (x, z arrays of same size)

\[ r = y + z \]  
(r, y, z scalar) OR (r, y, z arrays of same size)
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

• write constraints
  - each operation or function call imposes certain “constraints”
  - incomparable types give rise to multiple valid configurations
Static Type Inference  
(Appears in McCosh’s Masters’ Thesis)

- write constraints
  - each operation or function call imposes certain “constraints”
  - incomparable types give rise to multiple valid configurations

- the problem is hard to solve in general
  - efficient solution possible under certain conditions
Static Type Inference
(Appears in McCosh’s Masters’ Thesis)

- write constraints
  - each operation or function call imposes certain “constraints”
  - incomparable types give rise to multiple valid configurations

- the problem is hard to solve in general
  - efficient solution possible under certain conditions

- reducing to the clique problem
  - a constraint defines a level
  - clauses in a constraint are nodes at that level
  - an edge whenever two clauses are “compatible”
  - a clique defines a valid type configuration
Limitations

- control join-points may result in too many configs
Limitations

• control join-points may result in too many configs

• array sizes defined by indexed expressions
  - assignment to \( a(i) \) can resize \( a \)

• control join-points ignored for array-sizes

• symbolic expressions may be unknown at compile time

• array sizes changing in a loop not handled
Size Grows in a Loop

function [A, F] = pisar (xt, sin_num)
...
mcos = [];
for n = 1:sin_num
    vcos = [];
    for i = 1:sin_num
        vcos = [vcos cos(n*w_est(i))];
    end
    mcos = [mcos; vcos]
end
...

Slice-hoisting: Simple Example

\begin{verbatim}
A = zeros(1, N);

y = ...  
A(y) = ... 

x = ... 
A(x) = ... 
\end{verbatim}
Slice-hoisting: Simple Example

\[
A = \text{zeros}(1, N);
\]

\[
\sigma A = \langle N \rangle
\]

\[
y = \ldots
\]

\[
A(y) = \ldots
\]

\[
\sigma A = \max(\sigma A, \langle y \rangle)
\]

\[
x = \ldots
\]

\[
A(x) = \ldots
\]

\[
\sigma A = \max(\sigma A, \langle x \rangle)
\]
Slice- hoisting: Simple Example

\[
A_1 = \text{zeros}(1, N);
\]
\[
\sigma_1^{A_1} = \langle N \rangle
\]
\[
y_1 = \ldots
\]
\[
A_1(y_1) = \ldots
\]
\[
\sigma_2^{A_1} = \max(\sigma_1^{A_1}, \langle y_1 \rangle)
\]
\[
x_1 = \ldots
\]
\[
A_1(x_1) = \ldots
\]
\[
\sigma_3^{A_1} = \max(\sigma_2^{A_1}, \langle x_1 \rangle)
\]
Slice-hoisting: Simple Example

\[ A_1 = \text{zeros}(1, N); \]
\[ \Rightarrow \sigma_1^{A_1} = \langle N \rangle \]
\[ \Rightarrow y_1 = \ldots \]
\[ A_1(y_1) = \ldots \]
\[ \Rightarrow \sigma_2^{A_1} = \max(\sigma_1^{A_1}, \langle y_1 \rangle) \]
\[ \Rightarrow x_1 = \ldots \]
\[ A_1(x_1) = \ldots \]
\[ \Rightarrow \sigma_3^{A_1} = \max(\sigma_2^{A_1}, \langle x_1 \rangle) \]
Slice-hoisting: Simple Example

\[ \Rightarrow \sigma_1^{A_1} = \langle N \rangle \]
\[ \Rightarrow y_1 = \ldots \]
\[ \Rightarrow \sigma_2^{A_1} = \max(\sigma_1^{A_1}, \langle y_1 \rangle) \]
\[ \Rightarrow x_1 = \ldots \]
\[ \Rightarrow \sigma_3^{A_1} = \max(\sigma_2^{A_1}, \langle x_1 \rangle) \]
\[ \text{allocate}(A_1, \sigma_3^{A_1}); \]
\[ A_1 = \text{zeros}(1, N); \]
\[ A_1(y_1) = \ldots \]
\[ A_1(x_1) = \ldots \]
Slice-hoisting: Steps

- insert $\sigma$ statements
- do SSA conversion
- identify the slice involved in computing the $\sigma$ values
- hoist the slice before the first use of the array
Slice-hoisting: Loop

\[ A(x) = \ldots \]

\begin{verbatim}
for i = 1:N
    \ldots
    A = [A f(i)];
end
\end{verbatim}
Slice-hoisting: Loop

\[ A(x) = \ldots \]
\[ \sigma A = <x> \]
\[ \text{for } i = 1:N \]
\[ \ldots \]
\[ A = [A \ f(i)]; \]
\[ \sigma A = \sigma A + <1> \]
\[ \text{end} \]

- add \( \sigma \) statements
Slice-hoisting: Loop

\[ A_1(x_1) = \ldots \]

\[ \sigma_1^{A_1} = \langle x_1 \rangle \]

for \( i_1 = 1:N \)

\[ \ldots \]

\[ \sigma_2^{A_1} = \phi(\sigma_1^{A_1}, \sigma_3^{A_1}) \]

\[ A_1 = [A_1 \ f(i_1)]; \]

\[ \sigma_3^{A_1} = \sigma_2^{A_1} + \langle 1 \rangle \]

end

- add \( \sigma \) statements
- do SSA
Slice-hoisting: Loop

\[ A_1(x_1) = \ldots \]

\[ \Rightarrow \sigma_1^{A_1} = <x_1> \]

\[ \Rightarrow \text{for } i_1 = 1:N \]

\[ \ldots \]

\[ \Rightarrow \sigma_2^{A_1} = \phi(\sigma_1^{A_1}, \sigma_3^{A_1}) \]

\[ A_1 = [A_1 \ f(i_1)]; \]

\[ \Rightarrow \sigma_3^{A_1} = \sigma_2^{A_1} + <1> \]

\[ \Rightarrow \text{end} \]

- add \( \sigma \) statements
- do SSA
- identify slice
Slice-hoisting: Loop

\[ \sigma_1^{A_1} = \langle x_1 \rangle \]
\[ \text{for } i_1 = 1:N \]
\[ \sigma_2^{A_1} = \phi(\sigma_1^{A_1}, \sigma_3^{A_1}) \]
\[ \sigma_3^{A_1} = \sigma_2^{A_1} + \langle 1 \rangle \]
\[ \text{end} \]

\begin{verbatim}
allocate(A_1, \sigma_3^{A_1});
A_1(x_1) = ... 
for i_1 = 1:N
    ...
    A_1 = [A_1 f(i_1)];
end
\end{verbatim}

- add \( \sigma \) statements
- do SSA
- identify slice
- hoist the slice
Type-based Specialization

jakes: Type-specialized FORTRAN vs MATLAB

- MATLAB 6.x
- MATLAB 5.3
- FORTRAN

PhD Thesis Defense: Telescoping MATLAB for DSP Applications

July 10, 2003
Precision of Static Inference

![Bar chart showing the comparison between configurations with and without annotations. The x-axis represents different functions: acf, art. Q, fft, fourier by jump, and huffcode. The y-axis represents the number of configs. The chart indicates that the number of configs without annotations is significantly lower than with annotations, especially for the 'fourier by jump' function.](chart.png)
Relevant Optimizations

“It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.”

—Sir Arthur Conan Doyle in a A Scandal in Bohemia
Identifying and Discovering

- study of DSP applications
  - real life code from the ECE department
- identified high-payoff well-known optimization techniques
- discovered two novel optimizations
  - procedure strength reduction
  - procedure vectorization
High-payoff Optimizations

- vectorization
  - 33 times speedup in one case!
- common subexpression elimination
- beating and dragging along
- constant propagation
High-payoff Optimizations

- vectorization
  - 33 times speedup in one case!

- common subexpression elimination

- beating and dragging along

- constant propagation

- library identities
  - single call replaces a sequence

- value of library annotations
Procedure Strength Reduction

\[ \text{Procedure:}\]

\[
\text{Strength Reduction for } i=1:N \\
\ldots \\
\text{for } i=1:N \\
\ldots
\]

\[\text{end}\]

\[\text{f}(c_1, c_2, i, c_3);\]

\[\text{init}(c_1, c_2, c_3);\]

\[\text{for } i=1:N \\
\ldots \\
\text{f}_{\text{iter}}(i); \\
\ldots
\]

\[\text{end}\]
Procedure Strength Reduction

for i = 1:N
    ...
    f (c_1, c_2, i, c_3);
    ...
end
Procedure Strength Reduction

\begin{align*}
\text{for } i = 1:N \\
\quad &\ldots \\
\quad &f (c_1, c_2, i, c_3); \\
\quad &\ldots \\
\text{end}
\end{align*}

\begin{align*}
\text{f_init} (c_1, c_2, c_3); \\
\text{for } i = 1:N \\
\quad &\ldots \\
\quad &f\_\text{iter} (i); \\
\quad &\ldots \\
\text{end}
\end{align*}
Applying to ctss

speedups for top-level procedures in ctss relative to unoptimized

distribution of the total execution time among top-level procedures in ctss

PhD Thesis Defense: Telescoping MATLAB for DSP Applications

July 10, 2003
Applying to sML_chan_est

![Bar chart showing execution time (seconds) for different methods: original (per iteration), init call, init (with preallocation), and iterative call.](chart.png)
Effect of \texttt{mcc} Compilation

![Graph showing execution time for interpreted, compiled, and stand-alone modes with original and optimized versions.]
More on Strength Reduction

- procedure strength reduction somewhat different from operator strength reduction
  - could be similar if the `iter` component provided

- automatic differentiation is a more powerful approach that matches procedure strength reduction
  - more work needed to utilize automatic diff. for optimizing loops
Procedure Vectorization
Procedure Vectorization

\begin{verbatim}
for i = 1:N
    f (c_1, c_2, i, A[i]);
end
...
function f (a_1, a_2, a_3, a_4)
    \textless body of } f \text{ \textgreater
\end{verbatim}
for $i = 1:N$
    $f(c_1, c_2, i, A[i]);$
end

... 

function $f(a_1, a_2, a_3, a_4)$
    <$body of f$>
end

function $f_{vect}(c_1, c_2, [1:N], A)$
... 
function $f_{vect}(a_1, a_2, a_3, a_4)$
    for $i = 1:N$
        <$body of f$>
    end
Applying to jakes

![Graph showing speedup for 100 iterations]

- **Normalized Original**
- **Optimized**

**PhD Thesis Defense: Telescoping MATLAB for DSP Applications**

July 10, 2003
Overall Architecture

Parser and Front-End

Type Infer. Engine

Spl’n Engine

Code Gen.

Annot.

Lib.

Opt. Specs

C

PhD Thesis Defense: Telescoping MATLAB for DSP Applications

July 10, 2003
Contributions
Contributions

- validation of the telescoping languages strategy
  - the library compiler component
Contributions

• validation of the telescoping languages strategy
  - the library compiler component

• type-based specialization
  - $NP$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference
Contributions

- validation of the telescoping languages strategy
  - the library compiler component

- type-based specialization
  - \( \mathcal{NP} \)-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference

- identification of relevant optimizations

Contributions

- validation of the telescoping languages strategy
  - the library compiler component

- type-based specialization
  - \( \mathcal{NP} \)-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference

- identification of relevant optimizations

- discovery of two new optimizations
  - procedure strength reduction and procedure vectorization
Contributions

• validation of the telescoping languages strategy
  - the library compiler component

• type-based specialization
  - $NP$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference

• identification of relevant optimizations

• discovery of two new optimizations
  - procedure strength reduction and procedure vectorization

• infrastructure development
  - a novel compiler architecture
Contributions: Publications


- Arun Chauhan and Ken Kennedy. Slice-hoisting for dynamic size-inference in MATLAB. *In writing*.

Future Directions

- high-level reasoning
- time-bound compilation and AI techniques
- dynamic compilation and the grid
- automatic parallelization
- automatic differentiation
- other domains