SPL: A Language and Compiler for DSP Algorithms

Jianxin Xiong¹, Jeremy Johnson²
Robert Johnson³, David Padua¹

¹Computer Science, University of Illinois at Urbana-Champaign
²Mathematics and Computer Science, Drexel University
³MathStar Inc

http://polaris.cs.uiuc.edu/~jxiong/spl
Overview

- SPL: A domain specific language
  - DSP core algorithms
  - Matrix factorization

- SPL Compiler:
  - SPL $\Rightarrow$ Fortran/C programs
  - Efficient implementation

- Part of SPIRAL(www.ece.cmu.edu/~spiral):
  - Adaptive framework for optimizing DSP libraries
  - Search over different SPL formulas using SPL compiler.
Outline

- Motivation
- Mathematical formulation of DSP algorithms
- SPL Language
- SPL Compiler
- Performance Evaluation
- Conclusion
Motivation

- What affects the performance?
  - Architecture features:
    - pipeline, FU, cache, …
  - Compiler:
    - Ability to take advantage of architecture features
    - Ability to handle large / complicated programs

- Ideal compiler
  - Perform perfect optimization based on the architecture
  - Practical compilers have limitations
Motivation (continue)

- **Manual Performance Tuning**
  - Modify the source based on profiling information
  - Requires knowledge about the architecture features
  - Requires considerable work
  - The performance is not portable

- **Automatic performance tuning?**
  - Very difficult for general programs
  - DSP core algorithms: SPIRAL.
SPIRAL Framework

- DSP Transform
- Formula Generator
- SPL Formulae
- SPL Compiler
- C/FORTRAN Programs
- Performance Evaluation
- Search Engine
- Architecture
- DSP Libraries
Fast DSP Algorithms as Matrix Factorizations

- A DSP Transform:
  - \( y = Mx \Rightarrow y = M_1M_2...M_k x \)
- Example: n-point DFT \( y = F_n x \)

\[
F_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & -1 & i \\
\end{bmatrix}
\]
Tensor Product

- A linear algebra operation for representing repetitive matrix structures

\[ \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}_{m \times n} \otimes \begin{bmatrix} B \\ \vdots \\ B \end{bmatrix}_{n \times n'} = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}_{mm' \times nn'} \]

- Loop

\[ I \otimes B = \begin{bmatrix} B \\ \vdots \\ B \end{bmatrix} \]
Tensor Product (continue)

- Vector operations

\[ A \otimes I = \left[ \begin{array}{ccc} a_{11} & \cdots & \cdots \\ \vdots & a_{11} & \vdots \\ a_{m1} & \cdots & a_{m1} \end{array} \right] \cdots \left[ \begin{array}{ccc} a_{1n} & \cdots & \cdots \\ \vdots & a_{1n} & \vdots \\ a_{mn} & \cdots & a_{mn} \end{array} \right] \]
Rules for Recursive Factorization

- Cooley-Tukey factorization for DFT

\[ F_{rs} = (F_r \otimes I_s) T_{rs}^{r_s} (I_r \otimes F_s) L_{r}^{rs} \]

- General K-way factorization for DFT

\[ F_n = \prod_{i=1}^{k} \left[ (I_{n_i-} \otimes F_{n_i} \otimes I_{n_i+}) (I_{n_i-} \otimes T_{n_i+}^{n_i,n_i+}) \right] \cdot \prod_{i=k}^{1} (I_{n_i-} \otimes L_{n_i}^{n_i,n_i+}) \]

where \( n = n_1 \ldots n_k, \quad n_i- = n_1 \ldots n_{i-1}, \quad n_i+ = n_{i+1} \ldots n_k \)
Formulas

- Variations of DFT\((8)\)

\[
F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes F_2) L_2^8
\]

\[
F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes ((F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4)) L_2^8
\]

\[
F_8 = (F_2 \otimes I_4) T_4^8 (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2) R_8
\]
The SPL Language

- Domain-specific programming language for describing matrix factorizations

\[ F_4 = (F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4 \]

(matrix operations

primitives: parameterized special matrices)
SPL In A Nut-shell

- SPL expressions
  - General matrices
    - (matrix \((a_{11}...a_{1n}) \ldots (a_{m1} \ldots a_{mn})\))
    - (diagonal \((a_{11}...a_{nn})\))
    - (sparse \((i_1 j_1 a_1) \ldots (i_k j_k a_k)\))
  - Parameterized special matrices
    - \((I \ n), (L \ mn \ n), (T \ mn \ n), (F \ n)\)
  - Matrix operations
    - (compose \(A_1 \ldots A_k\) )
    - (tensor \(A_1 \ldots A_k\) )
    - (direct_sum \(A_1 \ldots A_k\) )
    - \(A \oplus B = \text{diag}(A,B)\)

- Others: definitions, directives, template, comments
A Simple SPL Program

; This is a simple SPL program
(define A (matrix(1 2)(2 1)))
(define B (diagonal(3 3)))
#subname simple
(tensor (I 2)(compose A B))
;; This is an invisible comment
The SPL Compiler

- Parsing
  - SPL Formula
  - Symbol Definition
  - Template Definition
- Abstract Syntax Tree
  - Symbol Table
  - Template Table
- Intermediate Code Generation
  - I-Code
- Intermediate Code Restructuring
  - I-Code
- Optimization
  - I-Code
- Target Code Generation
  - FORTRAN, C
Template Based Intermediate Code Generation

- Why use template?
  - User-defined semantics
  - Language extension
  - Compiler extension without modifying the compiler
  - Be integrated into the search space

- Structure of a template
  - Pattern, condition, code

- Template match
  - Generate I-code from matching template
  - Template matching is a recursive procedure
I-Code

- I-code is the intermediate code of the SPL compiler
- Internally I-code is four-tuples
  - \(<op, src1, src2, dest>\)
- The external representation of I-code
  - Fortran-like
  - Used in template
(template
  (F n)[ n >= 1 ]
  ( do i=0,n-1
    y(i)=0
    do j=0,n-1
      y(i)=y(i)+W(n,i*j)*x(j)
    end
  end ))
(F 2) matches pattern (F n) and assigns 2 to n. Because n=2 satisfies the condition n>=1, the following i-code is generated from the template:

\[
\begin{align*}
&\text{do } i = 0,1 \\
&\quad y(i) = 0 \\
&\text{do } j = 0,1 \\
&\quad \quad y(i) = y(i) + W(2, i*j) * x(j) \\
&\text{end} \\
&\text{end}
\end{align*}
\]

Y(0) = x(0) + x(1)
y(1) = x(0) - x(1)

Unrolling & Optimization
Define A Primitive

(primitive J)
(template
  (J n)
  [ n >= 1 ]
  ( do i=0,n-1
    y(i) = x(n-1-i)
  end ))

\[ J_n = \begin{bmatrix}
  1 \\
  \vdots \\
  1
\end{bmatrix}_{n \times n} \]
Define An Operation

(operation rcompose)
(template
  (rcompose A B)
  [ B.nx == A.ny ]
  ( t = A(x)
    y = B(t)))

\[ y = (A \circ B)x \]
\[ \equiv t = Ax \]
\[ y = Bt \]
Compound Template Matching

(rcompose (J 2) (F 2))
(rcompose A B)

\[ t = x \]
\[ y = t \]

\[ y(0) = x(1) + x(0) \]
\[ y(1) = x(1) - x(0) \]

\[ t(0) = x(1) \]
\[ t(1) = x(0) \]

\[ y(0) = t(0) + t(1) \]
\[ y(1) = t(0) - t(1) \]
Intermediate Code Restructuring

- Loop unrolling
  - Degree of unrolling can be controlled globally or case by case

- Scalar function evaluation
  - Replace scalar functions with constant value or array access

- Type conversion
  - Type of input data: real or complex
  - Type of arithmetic: real or complex
  - Same SPL formula, different C/Fortran programs
Optimizations

- Low-level optimizations:
  - Instruction scheduling, register allocation, instruction selection, …
  - Leave them to the native compiler

- Basic high-level optimizations:
  - Constant folding, copy propagation, CSE, dead code elimination,…
  - The native compiler is supposed to do the dirty work, but not enough.

- High-level scheduling, loop transformations:
  - Formula transformation
  - Integrated into the search space
Basic Optimizations (FFT, N=2^5, Ultra5)
Basic Optimizations (FFT, N=2^5, Origin200)
Basic Optimizations (FFT, N=2^5, PC)
Performance Evaluation

- Platforms: Ultra5, Origin 200, PC
- Small-size FFT ($2^1$ to $2^6$)
  - Straight-line code
  - K-way factorization
  - Dynamic programming
- Large-size FFT ($2^7$ to $2^{20}$)
  - Loop code
  - Binary right-most factorization
  - Dynamic programming
- Accuracy, memory requirement
FFT W

- A FFT package
  - Codelet: optimized straight-line code for small-size FFTs
  - Plan: factorization tree
  - Use dynamic programming to find the plan
  - Make recursive function calls to the codelet according to the plan
- Measure and estimate
FFT Performance ($N=2^1$ to $2^6$, Ultra5)
FFT Performance ($N=2^1$ to $2^6$, Origin200)
FFT Performance (N=2^1 to 2^6, PC)
FFT Performance (N=2^7 to 2^{20}, Ultra5)
FFT Performance (N=\(2^7\) to \(2^{20}\), Origin200)
FFT Performance (N=2^7 to 2^{20}, PC)
FFT Accuracy (N=2^1 to 2^{18})
FFT Memory Utilization (N=2^7 to 2^{20})

![Graph showing FFT Memory Utilization with different scaling sizes and memory usage in MB.](image)
Conclusion

• The SPL compiler is capable of producing efficient code on a variety of platforms.
• The standard optimizations carried out by the SPL compiler are necessary to get good performance.
• The template mechanism makes the SPL language and the SPL compiler highly extensible
## Related Work

<table>
<thead>
<tr>
<th>Domain</th>
<th>Code Generator</th>
<th>Tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFTW</td>
<td>FFT</td>
<td>Fix algorithms</td>
</tr>
<tr>
<td>WHT Package</td>
<td>WHT</td>
<td>Built-in</td>
</tr>
<tr>
<td>EXTENT</td>
<td>Block recursive</td>
<td>Built-in</td>
</tr>
<tr>
<td>ATLAS</td>
<td>BLAS</td>
<td>Hand coded, Blocking, unrolling</td>
</tr>
<tr>
<td>PHiPAC</td>
<td>BLAS</td>
<td>Hand coded</td>
</tr>
<tr>
<td>Iterative Compilation</td>
<td>Compiler option</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Performance Evaluation: Platforms

- **Ultra5**
  - Solaris 7, Sun Workshop 5.0
  - 333MHz UltraSPARC Iii, 128MB, 16KB/16KB/2MB

- **Origin 200**
  - IRIX64 6.5, MIPSpro 7.3.1.1m
  - 180MHz MIPS R10000, 384MB, 32KB/32KB/1MB

- **PC**
  - Linux kernel 2.2.18, egcs 1.1.2
  - 400MHz Pentium II, 256MB, 16K/16K/512KB