Automatic library generation

David Padua
University of Illinois at Urbana-Champaign
Libraries and Productivity

- Libraries help productivity.
- But not always.
  - Not all algorithms implemented.
  - Not all data structures.
- In any case, much effort goes into highly-tuned libraries.
- Automatic generation of libraries libraries would
  - Reduce cost of developing libraries
  - For a fixed cost, enable a wider range of implementations and thus make libraries more usable.
An Illustration based on MATLAB of the effect of libraries on performance
Compilers vs. Libraries in Sorting

IBM Power3

IBM Power4

~2X

Execution Time (cycle per key)
Standard Deviation

QuickSort →
IBM ESSL →
Compilers versus libraries in DFT

Vendor library (hand-tuned assembly) but also FFTW (adaptable library) and SPIRAL (generated code) vs reasonable implementation (Numerical recipes, GNU scientific library)
Compilers vs. Libraries in Matrix-Matrix Multiplication (MMM)
Library Generators

• Automatically generate highly efficient libraries for a class platforms.

• No need to manually tune the library to the architectural characteristics of a new machine.
Library Generators (Cont.)

• Examples:
  – In linear algebra: ATLAS, PhiPAC
  – In signal processing: FFTW, SPIRAL

• Library generators usually handle a fixed set of algorithms.

• Exception: SPIRAL accepts formulas and rewriting rules as input.
Library Generators (Cont.)

• At installation time, LGs apply empirical optimization.
  – That is, search for the best version in a set of different implementations
  – Number of versions astronomical. Heuristics are needed.
Library Generators (Cont.)

• LGs must output C code for portability.
• Uneven quality of compilers =>
  – Need for source-to-source optimizers
  – Or incorporate in search space variations introduced by optimizing compilers.
Library Generators (Cont.)

Algorithm description

Generator

C function

Source-to-source optimizer

C function

Native compiler

Object code

Execution

Final C function

performance
Important research issues

• Reduction of the search space with minimal impact on performance
• Adaptation to the input data (not needed for dense linear algebra)
• More flexible of generators
  – algorithms
  – data structures
  – classes of target machines
• Tools to build library generators.
Library generators and compilers

- LGs are a good yardstick for compilers
- Library generators use compilers.
- Compilers could use library generator techniques to optimize libraries in context.
- Search strategies could help design better compilers -
  - Optimization strategy: Most important open problem in compilers.
Organization of a library generation system

- **High Level Specification (Domain Specific Language (DSL))**
  - **Signal Processing Formula**
    - **Parameterization for Signal Processing**
  - **Linear Algebra Algorithm in Functional Language Notation**
    - **Parameterization for Linear Algebra**
  - **Parameterization for Program Generator for Sorting**

- **Selection Strategy**
  - **Backend Compiler**
    - **X Code with Search Directives**
      - **Reflective Optimization**
        - **Run**
          - **Executable**
            - **Run**
Three library generation projects

1. Spiral and the impact of compilers
2. ATLAS and analytical model
3. Sorting and adapting to the input
Spiral: A code generator for digital signal processing transforms

Joint work with:
- Jose Moura (CMU),
- Markus Pueschel (CMU),
- Manuela Veloso (CMU),
- Jeremy Johnson (Drexel)
SPIRAL

• The approach:
  - Mathematical formulation of signal processing algorithms
  - Automatically generate algorithm versions
  - A generalization of the well-known FFTW
  - Use compiler technique to translate formulas into implementations
  - Adapt to the target platform by searching for the optimal version
DSP Algorithms: Example 4-point DFT

Cooley/Tukey FFT (size 4):

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

Fourier transform

Diagonal matrix (twiddles)

\[
DFT_4 = (DFT_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes DFT_2) \cdot L_2^4
\]

Kronecker product

Identity

Permutation

\[
\rightarrow \text{ product of structured sparse matrices}
\]

\[
\rightarrow \text{ mathematical notation}
\]
Fast DSP Algorithms As Matrix Factorizations

- Computing $y = F_4 x$ is carried out as:
  
  $t_1 = A_4 x$ (permutation)
  
  $t_2 = A_3 t_1$ (two $F_2$'s)
  
  $t_3 = A_2 t_2$ (diagonal scaling)
  
  $y = A_1 t_3$ (two $F_2$'s)

- The cost is reduced because $A_1, A_2, A_3$ and $A_4$ are structured sparse matrices.
General Tensor Product Formulation

Theorem

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

\( T_{rs}^s \) is a diagonal matrix

\( L_{rs}^r \) is a stride permutation

Example

\[ F_4 = (F_2 \otimes I_2) T_4^4 (I_2 \otimes F_2) L_2^4 \]

\[
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & -1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Factorization Trees

Different computation order
Different data access pattern

Different performance
The SPIRAL System

DSP Transform

Formula Generator

SPL Program

SPL Compiler

C/FORTRAN Programs

Search Engine

Target machine

Performance Evaluation

DSP Library

C/FORTRAN Programs

Target machine
SPL Compiler

SPL Formula → Template Definition

Parsing

Abstract Syntax Tree → Template Table

Intermediate Code Generation

I-Code

Intermediate Code Restructuring

I-Code

Optimization

I-Code

Target Code Generation

FORTRAN, C
Optimizations

- High-level scheduling
- Loop transformation

- High-level optimizations
  - Constant folding
  - Copy propagation
  - CSE
  - Dead code elimination

- Low-level optimizations
  - Instruction scheduling
  - Register allocation
Basic Optimizations

(FFT, N=2^5, SPARC, f77 -fast -O5)
Basic Optimizations
(FFT, N=2^5, MIPS, f77 -O3)

![Graph showing different SPL formulas for FFT (N=32)]
Basic Optimizations
(FFT, N=2^5, PII, g77 -O6 -malign-double)
Overall performance
An analytical model for ATLAS

Joint work with
Keshav Pingali (Cornell)
Gerald DeJong
Maria Garzaran
ATLAS

• ATLAS = Automated Tuned Linear Algebra Software, developed by R. Clint Whaley, Antoine Petite and Jack Dongarra, at the University of Tennessee.

• ATLAS uses empirical search to automatically generate highly-tuned Basic Linear Algebra Libraries (BLAS).
  – Use search to adapt to the target machine
ATLAS Infrastructure

Detect Hardware Parameters
- L1Size
- NR
- MulAdd
- Latency

ATLAS Search Engine (MMSearch)
- NB
- MU, NU, KU
- xFetch
- MulAdd
- Latency

ATLAS MM Code Generator (MMCase)

Compile, Execute, Measure

MiniMMM Source

MFLOPS
Detecting Machine Parameters

- **Micro-benchmarks**
  - **L1Size**: L1 Data Cache size
    - Similar to Hennessy-Patterson book
  - **NR**: Number of registers
    - Use several FP temporaries repeatedly
  - **MulAdd**: Fused Multiply Add (FMA)
    - “c+=a*b” as opposed to “c+=t; t=a*b”
  - **Latency**: Latency of FP Multiplication
    - Needed for scheduling multiplies and adds in the absence of FMA
Compiler View

- ATLAS Code Generation

- Focus on MMM (as part of BLAS-3)
  - Very good reuse $O(N^2)$ data, $O(N^3)$ computation
  - No “real” dependencies (only input / reuse ones)
Adaptations/Optimizations

• Cache-level blocking (tiling)
  – Atlas blocks only for L1 cache
• Register-level blocking
  – Highest level of memory hierarchy
  – Important to hold array values in registers
• Software pipelining
  – Unroll and schedule operations
• Versioning
  – Dynamically decide which way to compute
Cache-level blocking (tiling)

- Tiling in ATLAS
  - Only square tiles (NBxNBxNB)
  - Working set of tile fits in L1
  - Tiles are usually copied to continuous storage
  - Special “clean-up” code generated for boundaries

- Mini-MMM

```c
for (int j = 0; j < NB; j++)
    for (int i = 0; i < NB; i++)
        for (int k = 0; k < NB; k++)
            C[i][j] += A[i][k] * B[k][j]
```

- **NB**: Optimization parameter
Register-level blocking

• Micro-MMM
  – MUx1 elements of A
  – 1xNU elements of B
  – MUxNU sub-matrix of C
  – MU*NU + MU + NU ≤ NR

• Mini-MMM revised
  ```java
  for (int j = 0; j < NB; j += NU)
      for (int i = 0; i < NB; i += MU)
          load C[i..i+MU-1, j..j+NU-1] into registers
          for (int k = 0; k < NB; k++)
              load A[i..i+MU-1,k] into registers
              load B[k,j..j+NU-1] into registers
              multiply A’s and B’s and add to C’s
              store C[i..i+MU-1, j..j+NU-1]
  ```

• Unroll K look KU times
• **MU, NU, KU**: optimization parameters
Scheduling

- FMA Present?
- Schedule Computation
  - Using Latency
- Schedule Memory Operations
  - Using FFetch, IFetch, NFetch
- Mini-MMM revised

```c
for (int j = 0; j < NB; j += NU)
    for (int i = 0; i < NB; i += MU)
        load C[i..i+MU-1, j..j+NU-1] into registers
    for (int k = 0; k < NB; k += KU)
        load A[i..i+MU-1,k] into registers
        load B[k,j..j+NU-1] into registers
        multiply A’s and B’s and add to C’s
        ...
        load A[i..i+MU-1,k+KU-1] into registers
        load B[k+KU-1,j..j+NU-1] into registers
        multiply A’s and B’s and add to C’s
        store C[i..i+MU-1, j..j+NU-1]
```

- Latency, xFetch: optimization parameters
Searching for Optimization Parameters

- ATLAS Search Engine

Multi-dimensional search problem
- Optimization parameters are independent variables
- MFLOPS is the dependent variable
- Function is implicit but can be repeatedly evaluated
Search Strategy

• Orthogonal Range Search
  – Optimize along one dimension at a time, using reference values for not-yet-optimized parameters
  – Not guaranteed to find optimal point
  – Input
    • Order in which dimensions are optimized
      – NB, MU & NU, KU, xFetch, Latency
    • Interval in which search is done in each dimension
      \[ 16 \leq NB \leq \min(\sqrt{L1Size}, 80) \]
      – For NB it is [ , step 4]
    • Reference values for not-yet-optimized dimensions
      – Reference values for KU during NB search are 1 and NB
Modeling for Optimization Parameters

• Our Modeling Engine

\[ NU + MU + MU + \text{Latency} \leq NR \]

• Optimization parameters
  – NB: Hierarchy of Models (later)
  – MU, NU:
  – KU: maximize subject to L1 Instruction Cache
  – Latency, MulAdd: from hardware parameters
  – xFetch: set to 2
Modeling for Tile Size (NB)

- Models of increasing complexity
  - \(3 \times NB^2 \leq C\)
    - Whole work-set fits in L1
  - \(NB^2 + NB + 1 \leq C\)
    - Fully Associative
    - Optimal Replacement
    - Line Size: 1 word
    \[
    \left\lceil \frac{NB^2}{B} \right\rceil + \left\lceil \frac{NB}{B} \right\rceil + 1 \leq \frac{C}{B} \quad \text{or} \quad \left\lceil \frac{NB^2}{B} \right\rceil + NB + 1 \leq \frac{C}{B}
    \]
    - Line Size > 1 word
    \[
    \left\lceil \frac{NB^2}{B} \right\rceil + 2\left\lceil \frac{NB}{B} \right\rceil + \left(\left\lceil \frac{NB}{B} \right\rceil + 1\right) \leq \frac{C}{B} \quad \text{or} \quad \left\lceil \frac{NB^2}{B} \right\rceil + 3NB + 1 \leq \frac{C}{B}
    \]
  - LRU Replacement
Experiments

• Architectures:
  – SGI R12000, 270MHz
  – Sun UltraSPARC III, 900MHz
  – Intel Pentium III, 550MHz

• Measure
  – Mini-MMM performance
  – Complete MMM performance
  – Sensitivity to variations on parameters
MiniMMM Performance

• SGI
  – ATLAS: 457 MFLOPS
  – Model: 453 MFLOPS
  – Difference: 1%

• Sun
  – ATLAS: 1287 MFLOPS
  – Model: 1052 MFLOPS
  – Difference: 20%

• Intel
  – ATLAS: 394 MFLOPS
  – Model: 384 MFLOPS
  – Difference: 2%
MMM Performance

- SGI
- Sun
- Intel

BLAS, COMPILED, ATLAS, MODEL
Sensitivity to NB and Latency on Sun

- Tile Size (NB)
- Latency

- MU & NU, KU, Latency, xFetch for all architectures
Sensitivity to NB on SGI

Tile Size (B: Best, A: ATLAS, M: Model)

MFLOPS

3*NB^2 \leq C

NB^2 + NB + 1 \leq C
Sorting

Joint work with
Maria Garzaran
Xiaoming Li
ESSL on Power3

IBM Power3

Execution Time (Cycle. per key)

Standard Deviation

Quicksort

CC-radix

IBM ESSL
ESSL on Power4

IBM Power4

Execution Time (Cycle per key)

Standard Deviation

Quicksort

CC-radix

IBM ESSL
Motivation

• No universally best sorting algorithm

• Can we automatically GENERATE and tune sorting algorithms for each platform?

• Performance of sorting depends not only on the platform but also on the input characteristics.
A first strategy: Algorithm Selection

- Select the best algorithm from Quicksort, Multiway Merge Sort and CC-radix.

- Relevant input characteristics: number of keys, entropy vector.
Algorithm Selection

IBM Power3

Execution Time (Cycle. per key)

Standard Deviation

Quicksort  CC-radix  Multi-way Merge  IBM ESSL  Adaptive Sort
A better Solution

• We can use different algorithms for different partitions

• Build Composite Sorting algorithms
  – Identify primitives from the sorting algorithms
  – Design a general method to select an appropriate sorting primitive at runtime
  – Design a mechanism to combine the primitives and the selection methods to generate the composite sorting algorithm
Sorting Primitives

• Divide-by-Value
  – A step in Quicksort
  – Select one or multiple pivots and sort the input array around these pivots
  – Parameter: number of pivots

• Divide-by-Position (DP)
  – Divide input into same-size sub-partitions
  – Use heap to merge the multiple sorted sub-partitions
  – Parameters: size of sub-partitions, fan-out and size of the heap
Sorting Primitives

• Divide-by-Radix (DR)
  – Non-comparison based sorting algorithm
  – Parameter: radix (r bits)
Selection Primitives

- Branch-by-Size
- Branch-by-Entropy
  - Parameter: number of branches, threshold vector of the branches
Leaf Primitives

- When the size of a partition is small, we stick to one algorithm to sort the partition fully.

- Two methods are used in the cleanup operation
  - Quicksort
  - CC-Radix
Composite Sorting Algorithms

- Composite sorting algorithms are built with these primitives.
- Algorithms are represented as trees.
Performance of Classifier Sorting

- Power3
Power4

IBM Power4

![Graph showing execution time (cycle per key) vs standard deviation for different algorithms: C++ STL, Gene Sort, XSort, IBM ESSL. The graph demonstrates the performance variation across different standard deviations.]