Can We Teach Computers To Write Fast Libraries?

Markus Püschel

*Collaborators:*
Franz Franchetti
Yevgen Voronenko

Electrical and Computer Engineering
Carnegie Mellon University

This work was supported by DARPA DESA program, NSF-NGS/ITR, NSF-ACR, NSF-CPA, and Intel

... and the Spiral team (only part shown)
Hiroshige, Temple in park near Osaka, ca. 1853-59
The Problem: Example DFT

- Standard desktop computer, vendor compiler, using optimization flags
- All implementations have roughly the same operations count (~ $4n\log_2(n)$)
- *Maybe the DFT is just difficult?*
The Problem: Example MMM

- Similar plots can be shown for all numerical kernels in linear algebra, signal processing, coding, crypto, ...

- What’s going on?

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)
Performance [Gflop/s]
Evolution of Processors (Intel)

Floating point peak performance [Mflop/s]  
CPU frequency [MHz]

Year

1993 1995 1997 1999 2001 2003 2005 2007

Pentium Pro  Pentium II  Pentium III  Pentium IV  Core 2 Duo  Core 2 Quad

single precision  double precision  CPU frequency

data: www.sandpile.org
Evolution of Processors (Intel)

Floating point peak performance [Mflop/s]
CPU frequency [MHz]

Year
1993 1995 1997 1999 2001 2003 2005 2007

Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Core 2 Duo
Core 2 Quad

free speedup

data: www.sandpile.org
Carnegie Mellon

Evolution of Processors (Intel)

Floating point peak performance [Mflop/s]
CPU frequency [MHz]

100,000
10,000
1,000
100
10

work required
free speedup

1993 1995 1997 1999 2001 2003 2005 2007
Year

data: www.sandpile.org

Era of parallelism

High performance library development becomes increasingly difficult
DFT Plot: Analysis

Discrete Fourier Transform (DFT) on 2 x Core 2 Duo 3 GHz
Gflop/s

- Multiple threads: 2x
- Vector instructions: 3x
- Memory hierarchy: 5x

High performance library development has become a nightmare
Current Solution

- **Legions** of programmers implement and optimize the same functionality for **every** platform and **whenever** a new platform comes out.
Better Solution: Automatic Performance Tuning

- Automate (parts of) the implementation or optimization

- Research efforts
  - Linear algebra: Phipac/ATLAS (UTK), Sparsity/Bebop (Berkeley), Flame (UT Austin)
  - Tensor computations (Ohio State)
  - PDE/finite elements: Fenics
  - Adaptive sorting (UIUC)
  - Fourier transform: FFTW (MIT)
  - Linear transforms: Spiral
  - … others
  - New compiler techniques

Promising new area but more work needed
In particular for parallelism …
Organization

- Spiral: Brief overview
- Parallelization in Spiral
- Results
- Conclusion

Vision Behind Spiral

Current

- Numerical problem
- C program
- algorithm selection
- implementation
- compilation
- Computing platform
- human effort

Future

- Numerical problem
- C program
- algorithm selection
- implementation
- compilation
- Computing platform
- automated

- Challenge: conquer the high abstraction level for complete automation

- C code a singularity: Compiler has no access to high level information
Spiral

- Library generator for linear transforms (DFT, DCT, DWT, filters, ...) and recently more ...

- Wide range of platforms supported: scalar, fixed point, vector, parallel, Verilog

- Research Goal: “Teach” computers to write fast libraries
  - Complete automation of implementation and optimization
  - Conquer the “high” algorithm level for automation

- When a new platform comes out:
  Regenerate a retuned library

- When a new platform paradigm comes out (e.g., vector or CMPs):
  Update the tool rather than rewriting the library

Intel has started to use Spiral to generate parts of their MKL/IPP library
How Spiral Works

**Spiral:** Complete automation of the implementation and optimization task

**Basic ideas:**
Declarative representation of algorithms

Rewriting systems to generate and optimize algorithms at a high level of abstraction

Problem specification (transform) → Algorithm Generation

Algorithm Generation → Algorithm Optimization

Algorithm Optimization → Implementation

Implementation → Code Optimization

Code Optimization → Compilation

Compilation → Compiler Optimizations

Compiler Optimizations → Fast executable

Search

controls

controls

performance
### Program Generation Interface

**Target platform for optimization:** 2x Intel Xeon 3.6 GHz with 2048K L2 cache

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>DCT2 (Discrete Cosine Transform, type 2)</td>
<td>The transform for which you want to request C code</td>
</tr>
<tr>
<td>Data type</td>
<td>double</td>
<td>The data type of input and output vector</td>
</tr>
<tr>
<td>Transform size</td>
<td>6</td>
<td>The size of the transform = the length of the input vector</td>
</tr>
<tr>
<td>Optimize for</td>
<td>runtime</td>
<td>What you want to optimize the code for</td>
</tr>
<tr>
<td>Search method</td>
<td>Dynamic Programming</td>
<td>The search method SPIRAL uses (Dynamic Programming is a good choice)</td>
</tr>
<tr>
<td>Compiler profile</td>
<td>gcc -O3</td>
<td>Compiler and compiler options used for compilation</td>
</tr>
</tbody>
</table>

[Generate code]

### Browse Archive

**Filter by Platform:** All Platforms Selected

**Filter by Transform:** All Transforms Selected

**Filter by Size:** All Sizes Selected

[Query Database]
What is a (Linear) Transform?

- Mathematically: Matrix-vector multiplication

\[ x \xrightarrow{\text{transform}} y = T \cdot x \]

- Example: Discrete Fourier transform (DFT)

\[ \text{DFT}_n = [e^{-2\pi i k\ell/n}]_{0 \leq k, \ell < n} \]
Cooley/Tukey fast Fourier transform (FFT):

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & j & -1 & -j \\
1 & -1 & 1 & -1 \\
1 & -j & -1 & j
\end{bmatrix} = 
\begin{bmatrix}
1 & 1 & \cdots & 1 \\
1 & 1 & \cdots & \cdots & 1 \\
1 & \cdots & 1 & \cdots & \cdots & 1 \\
1 & \cdots & \cdots & 1 & \cdots & 1
\end{bmatrix}
\]

Fourier transform

\[
\text{DFT}_4 \rightarrow (\text{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \text{DFT}_2) L_2^4
\]

- Algorithms are divide-and-conquer: Breakdown rules
- Mathematical, declarative representation: SPL (signal processing language)
- SPL describes the structure of the dataflow
Examples: Transforms (currently 55)

\[
\begin{align*}
\text{DCT-2}_n &= \left[ \cos(k(2\ell+1)\pi/2n) \right]_{0 \leq k, \ell < n'}, \\
\text{DCT-3}_n &= \text{DCT-2}_n^T \quad \text{(transpose),} \\
\text{DCT-4}_n &= \left[ \cos((2k+1)(2\ell+1)\pi/4n) \right]_{0 \leq k, \ell < n'}, \\
\text{IMDCT}_n &= \left[ \cos((2k+1)(2\ell+1+n)\pi/4n) \right]_{0 \leq k < 2n, 0 \leq \ell < n'} \\
\text{RDFT}_n &= [r_{k\ell}]_{0 \leq k, \ell < n}, \quad r_{k\ell} = \begin{cases} 
\cos\frac{2\pi k\ell}{n}, & k \leq \left\lfloor \frac{n}{2} \right\rfloor \\
-\sin\frac{2\pi k\ell}{n}, & k > \left\lfloor \frac{n}{2} \right\rfloor 
\end{cases}, \\
\text{WHT}_n &= \begin{bmatrix} \text{WHT}_{n/2} & \text{WHT}_{n/2} \\
\text{WHT}_{n/2} & -\text{WHT}_{n/2} \end{bmatrix}, \quad \text{WHT}_2 = \text{DFT}_2, \\
\text{DHT} &= \left[ \cos(2k\ell\pi/n) + \sin(2k\ell\pi/n) \right]_{0 \leq k, \ell < n}.
\end{align*}
\]
Examples: Breakdown Rules (currently $\approx 220$)

\[
\begin{align*}
\text{DFT}_n & \rightarrow (\text{DFT}_k \otimes I_m) \cdot T_m^n (I_k \otimes \text{DFT}_m) \cdot L_k^n, \quad n = km \\
\text{DFT}_n & \rightarrow P_n (DFT_k \otimes DFT_m) Q_n, \quad n = km, \quad \gcd(k, m) = 1 \\
\text{DFT}_p & \rightarrow R_p^T (I_1 \oplus \text{DFT}_{p-1}) D_p (I_1 \oplus \text{DFT}_{p-1}) R_p, \quad p \text{ prime} \\
\text{DCT-3}_n & \rightarrow (I_m \oplus J_m) \cdot L_m^n (\text{DCT-3}_m(1/4) \oplus \text{DCT-3}_m(3/4)) \\
& \cdot (F_2 \otimes I_m) \left[ I_m \begin{bmatrix} 0 & -J_m^{-1} \\ \frac{1}{\sqrt{2}} (I_1 \oplus 2 I_m) \end{bmatrix}, \quad n = 2m \right]
\end{align*}
\]

\[
\begin{align*}
\text{DCT-4}_n & \rightarrow S_n \text{DCT-2}_n \cdot \text{diag}_{0 \leq k < n} (1/(2 \cos((2k + 1)\pi/4n))) \\
\text{IMDCT}_2^{2m} & \rightarrow (J_m \oplus I_m \oplus I_m \oplus J_m) \sqrt{\frac{1}{m}} \left( \left[ \begin{array}{c} 1 \\ -1 \end{array} \right] \otimes I_m \right) \oplus \left( \left[ \begin{array}{c} -1 \\ -1 \end{array} \right] \otimes I_m \right) J_{2m} \text{DCT-4}_m
\end{align*}
\]

\[
\begin{align*}
\text{WHT}_2^{2k} & \rightarrow \prod_{i=1}^{t} (I_{2^{k_1+\cdots+k_{i-1}}} \otimes \text{WHT}_{2^{k_i}} \otimes I_{2^{k_{i+1}+\cdots+k_t}}), \quad k = k_1 + \cdots + k_t \\
\text{DFT}_2 & \rightarrow F_2 \\
\text{DCT-2}_2 & \rightarrow \text{diag}(1, 1/\sqrt{2}) F_2 \\
\text{DCT-4}_2 & \rightarrow J_2 R_{13\pi/8}
\end{align*}
\]

Base case rules

Combining these rules yields many algorithms for every given transform
Breakdown rules in Spiral:

- "Teaches" Spiral about existing algorithm knowledge (~200 journal papers)
- Includes many new ones (algebraic theory)

Cooley-Tukey FFT
# SPL to Sequential Code

<table>
<thead>
<tr>
<th>SPL construct</th>
<th>code</th>
</tr>
</thead>
</table>
| $y = (A_n B_n) x$ | $t[0:1:n-1] = B(x[0:1:n-1]);$  
| | $y[0:1:n-1] = A(t[0:1:n-1]);$ |
| $y = (I_m \otimes A_n) x$ | for (i=0; i<m; i++)  
| | $y[i*n:1:i*n+n-1] =$  
| | $A(x[i*n:1:i*n+n-1]);$ |
| $y = (A_m \otimes I_n) x$ | for (i=0; i<m; i++)  
| | $y[i*n:i+m-1] =$  
| | $A(x[i*n:i+m-1]);$ |
| $y = (\bigoplus_{i=0}^{m-1} A_n^i) x$ | for (i=0; i<m; i++)  
| | $y[i*n:1:i*n+n-1] =$  
| | $A(i, x[i*n:1:i*n+n-1]);$ |
| $y = D_{m,n} x$ | for (i=0; i<m*n; i++)  
| | $y[i] = D_{m,n}[i]*x[i];$ |
| $y = L_{m}^{mn} x$ | for (i=0; i<m; i++)  
| | for (j=0; j<n; j++)  
| | $y[i+m*j] = x[n*i+j];$ |

Example: tensor product

\[
I_m \otimes A_n = \begin{bmatrix} A_n & \cdots \\ \vdots \\ A_n \end{bmatrix}
\]

**Correct code:** easy  
**Fast code:** very difficult
Program Generation in Spiral (Sketched)

Transform
user specified

Fast algorithm
in SPL
many choices

\[ DFT_8 \]

\[ \sum (S_j DFT_2 G_j) \sum \left( \sum \left( S_{k,l} \text{diag}(t_{k,l}) DFT_2 G_l \right) \right) \sum \left( S_m \text{diag}(t_m) DFT_2 G_{k,m} \right) \]

void sub(double *y, double *x) {
    double f0, f1, f2, f3, f4, f7, f8, f10, f11;
    f0 = x[0] - x[3];
    f1 = x[0] + x[3];
    f2 = x[1] - x[2];
    f3 = x[1] + x[2];
    f4 = f1 - f3;
    y[0] = f1 + f3;
    y[2] = 0.7071067811865476 * f4;
    f7 = 0.9238795325112867 * f0;
    f8 = 0.3826834323650898 * f2;
    y[1] = f7 + f8;
    f10 = 0.3826834323650898 * f0;
    f11 = (-0.9238795325112867) * f2;
    y[3] = f10 + f11;
}

Optimization at all abstraction levels

parallelization
vectorization
loop optimizations
constant folding
scheduling

……
Organization

- Spiral: Brief overview
- Parallelization in Spiral
- Results
- Conclusion
SPL to Shared Memory Code: Basic Idea [SC 06]

- Governing construct: tensor product
  \[ y = (I_p \otimes A)x \]
  p-way embarrassingly parallel, load-balanced

- Problematic construct: permutations produce false sharing
  \[ y = L_4^8 x \]

**Task:** Rewrite formulas to extract tensor product + keep contiguous blocks
Step 1: Shared Memory Tags

- Identify crucial hardware parameters
  - Number of processors: \( p \)
  - Cache line size: \( \mu \)

- Introduce them as tags in SPL

\[
A_{\text{smp}(p,\mu)}
\]

This means: formula A is to be optimized for \( p \) processors and cache line size \( \mu \)
Step 2: Identify “Good” Formulas

- **Load balanced, avoiding false sharing**

\[
y = (I_p \otimes A)x \quad \text{with} \quad A \in \mathbb{C}^{m \mu \times m \mu}
\]

\[
y = \left(\bigoplus_{i=0}^{p-1} A_i\right)x \quad \text{with} \quad A_i \in \mathbb{C}^{m \mu \times m \mu}
\]

\[
y = (P \otimes I_\mu)x \quad \text{with} \quad P \text{ a permutation matrix}
\]

- **Tagged operators (no further rewriting necessary)**

\[
I_p \otimes\|\cdot\|, \quad \bigoplus_{i=0}^{p-1} \|A_i\|, \quad P \otimes I_\mu
\]

- **Definition:** A formula is **fully optimized** for \((p, \mu)\) if it is one of the above or of the form

\[
I_m \otimes A \quad \text{or} \quad AB
\]

where \(A\) and \(B\) are fully optimized.
Step 3: Identify Rewriting Rules

**Goal:** Transform formulas into fully optimized formulas

- Formulas rewritten, tags propagated
- There may be choices

\[
\begin{align*}
\frac{AB}{\text{smp}(p,\mu)} & \rightarrow \frac{A}{\text{smp}(p,\mu)} \frac{B}{\text{smp}(p,\mu)} \\
\frac{A_m \otimes I_n}{\text{smp}(p,\mu)} & \rightarrow \left( \frac{L_{m/p}^{mp} \otimes I_{n/p}}{\text{smp}(p,\mu)} \right) \left( \frac{I_p \otimes (A_m \otimes I_{n/p})}{\text{smp}(p,\mu)} \right) \left( \frac{L_{p}^{mp} \otimes I_{n/p}}{\text{smp}(p,\mu)} \right) \\
\frac{L_{m}^{mn}}{\text{smp}(p,\mu)} & \rightarrow \left\{ \begin{array}{c}
\frac{I_p \otimes L_{m/p}^{mn}}{\text{smp}(p,\mu)} \frac{L_{p}^{pn} \otimes I_{m/p}}{\text{smp}(p,\mu)} \\
\frac{L_{m}^{pm} \otimes I_{n/p}}{\text{smp}(p,\mu)} \frac{I_p \otimes L_{m}^{mn}}{\text{smp}(p,\mu)}
\end{array} \right\} \\
\frac{I_m \otimes A_n}{\text{smp}(p,\mu)} & \rightarrow I_p \otimes \left( I_{m/p} \otimes A_n \right) \\
\frac{P \otimes I_n}{\text{smp}(p,\mu)} & \rightarrow \left( P \otimes I_{n/\mu} \right) \overline{\otimes} I_{\mu}
\end{align*}
\]
Simple Rewriting Example

\[
A_m \otimes I_n
\]

Loop splitting + loop exchange

\[
(L_m^{mp} \otimes I_{n/p})(I_p \otimes \| (A_m \otimes I_{n/p}) \otimes L_p^{mp} \otimes I_{n/p})
\]

fully optimized

parallel for (i=0; i<p; i++)
    for (j=0; j<n/p; j++)
        y[i*n/p+j:n:i*n/p+j+m-1] = A(x[i*n/p+j:n:i*n/p+j+m-1]);
Parallelization by Rewriting

\[
\frac{DFT_{mn}}{\text{smp}(p,\mu)} \rightarrow \left( \frac{(DFT_m \otimes I_n) T_{n}^{mn} (I_m \otimes DFT_n) L_{m}^{mn}}{\text{smp}(p,\mu)} \right)
\]

\[
\ldots
\]

\[
\rightarrow \left( \frac{DFT_m \otimes I_n}{\text{smp}(p,\mu)} \right) T_n^{mn} \left( \frac{I_m \otimes DFT_n}{\text{smp}(p,\mu)} \right) L_m^{nm} \frac{\text{smp}(p,\mu)}{\text{smp}(p,\mu)}
\]

\[
\ldots
\]

\[
\rightarrow \left( (L_m^{mp} \otimes I_{n/p\mu}) \otimes \mu I_\mu \right) \left( I_p \otimes \| (DFT_m \otimes I_{n/p}) \right) \left( (L_p^{mp} \otimes I_{n/p\mu}) \otimes \mu I_\mu \right)
\]

\[
\left( \bigoplus_{i=0}^{p-1} T_n^{mn,i} \right) \left( I_p \otimes \| (I_m/p \otimes DFT_n) \right) \left( I_p \otimes \| L_m^{mn/p} \right) \left( (L_p^{pn} \otimes I_{m/p\mu}) \otimes \mu I_\mu \right)
\]

Fully optimized (load-balanced, no false sharing) in the sense of our definition
Same Approach for Other Parallel Paradigms

Message Passing: [ISPA 06]

Vectorization: [IPDPS 02, VecPar 06]

Cg/OpenGL for GPUs:

Verilog for FPGAs: [DAC 05]
Going Beyond Transforms

- **Transform =** linear operator with one vector input and one vector output

- **Key ideas:**
  - Generalize to (possibly nonlinear) operators with several inputs and several outputs
  - Generalize SPL (including tensor product) to OL (operator language)
  - Generalize rewriting systems for parallelizations

**Cooley-Tukey FFT in OL:**
\[ \text{DFT} \rightarrow (\text{DFT} \otimes \text{I}) \circ D \circ (\text{I} \otimes \text{DFT}) \circ \text{L}. \]

**Viterbi in OL:**
\[ \text{Vit} \rightarrow \pi \circ (\prod (\text{I} \otimes \text{V}) \circ (\text{L} \times \text{I})) \circ (\text{C} \times \text{C} \times \text{I}) \]

**Mat-Mat-Mult:**
\[ \text{MMM} \rightarrow \text{I} \otimes \text{MMM} \]
\[ \text{MMM} \rightarrow (\text{I} \otimes \text{L}) \circ (\text{MMM} \otimes \text{I}) \circ (\text{I} \times (\text{I} \otimes \text{L})) \]

- **We have first results beyond transforms!**
Organization

- Spiral overview
- Parallelization in Spiral
- Results
- Concluding remarks
All Spiral code shown is “push-button” generated from scratch

“click”
Benchmark: Vector and SMP

DFT (single precision): on 3 GHz 2 x Core 2 Extreme
performance [Gflop/s]

- Spiral 5.0 SPMD
- Spiral 5.0 sequential
- Intel IPP 5.0
- FFTW 3.2 alpha SMP
- FFTW 3.2 alpha sequential

Memory footprint < L1$ of 1 processor

4-way vectorized + up to 4-threaded + adapted to the memory hierarchy
DCT4, Multiples of 32: 4-way Vectorized

DCT (single precision) 2.66 GHz Core2 (4-way 32-bit SSE) performance [Gflop/s]

Less popular transform usually means no good code
Benchmark: Cell (1 processor = SPE)

DFT (single precision) on 3.2 GHz Cell BE (Single SPE)
performance [Gflop/s]

Generated using the simulator; run at Mercury (thanks to Robert Cooper)

Joint work with Th. Peter (ETH Zurich), S. Chellappa, M. Telgarsky, J. Moura (CMU)
Benchmark: GPU

WHT (single precision) on 3.6 GHz Pentium 4 with Nvidia 7900 GTX performance [Gflops/s]

Joint work with H. Shen, TU Denmark
Benchmark: FPGA

DFT 256 on Xilinx Virtex 2 Pro FPGA
inverse throughput (gap) [us]

- competitive with professional designs
- much larger set of performance/area trade-offs

Joint work with P. Milder, J. Hoe (CMU)
Benchmarks

kernels

- GEMM
- Viterbi
- filter
- DFT

platforms

- vector
- dual/quad core
- FPGA+CPU
- GPU
- FPGA

All Spiral code shown is “push-button” generated from scratch

“click”
Beyond Transforms: Viterbi Decoding

Viterbi decoding (8-bit) on 2.66 GHz Core 2 Duo

performance [Gbutterflies/s]

1 butterfly
= ~22 ops

Vectorized using practically the same rules as for DFT

Joint work with S. Chellappa, CMU

Karn: http://www.ka9q.net/code/fec/
Beyond Transforms: Matrix-Matrix-Multiply

MMM(square matrix real double) 2 x Core2Duo 3Ghz

performance [Gflop/s]

- **Spiral generated (pthreads + SSE)**
- **ATLAS contributed (pthreads+SSE)**
- **ATLAS generated (scalar)**
- **Goto (pthreads+SSE)**

All measurements with warm cache

ATLAS code is optimized for cold cache

work with F. de Mesmay, CMU
Organization

- Spiral overview
- Parallelization in Spiral
- Results
- Concluding remarks
Spiral Summary

- **Spiral: The computer implements and optimizes transforms**
  - Rigorous formal framework
  - Support for different platform paradigms (vector, SMP, GPU, FPGA, …)
  - Optimization at high abstraction level through rewriting + empirical search over alternatives
  - The generated code is often faster than human written code
  - We have extended the framework beyond transforms

- **What we have learned**
  - Declarative representation of algorithms (mathematical DSL)
  - Optimization at a high, “right” level of abstraction using rewriting
  - It makes sense to use math to represent and optimize math functionality
  - It makes sense to “teach” the computer algorithms and math (does not become obsolete)
  - Domain-specific is necessary to get fastest code
  - One needs techniques from different disciplines ….
Attacking the Parallel Library Challenge
Automation in High Performance Library Development

We Need to Work Together