Algorithms and Computation in Signal Processing

special topic course 18-799B
spring 2005
24th and 25th Lecture Apr. 07 and 12, 2005

Instructor: Markus Pueschel
TA: Srinivas Chellappa
Research Projects

- **Presentations last week of April (26th and 28th)**
  - We distribute the dates in the lecture on the 12th
  - Presentations 20 minutes + 5 minutes questions (~17-20 slides)

- **Research paper**
  - Due April 20th, the only thing that may be missing are some (but not all)
    experimental results
  - You’ll get feedback from me
  - Final version with feedback incorporated due one week after your presentation

- **Remarks**
  - Follow guide to benchmarking!
  - Try different sets of compiler flags to be sure
  - Do a cost analysis
Writing fast numerical code is a tough problem.
Moore’s Law

- Moore’s Law: exponential (x2 in ~18 months) increase number of transistors/chip

But everything has its price …
Moore’s Law: Consequences

- **Computers are very complex**
  - multilevel memory hierarchy
  - special instruction sets beyond standard C programming model
  - undocumented hardware optimizations

- **Consequences:**
  - Runtime depends only roughly on the operations
  - Runtime behavior is hard to understand
  - Compiler development can hardly keep track
  - **The best code (and algorithm) is platform-dependent**
  - It is very difficult to write really fast code

- **Computers evolve fast**
  - Highly tuned code becomes obsolete almost as fast as it as written

- **It’ll get rather worse: Multicoresystems**
Solution #1: Brute Force

- Thousands of programmers hand-write and hand-tune (assembly) code for the same numerical problems and for every platform and whenever a new platform comes out?

*Hmm…..
* (but it’s current practice)
Solution #2: New Approaches to Code Optimization and Code Creation

- **ATLAS**: Code generation/optimization for BLAS

- **SPARSITY/BeBop**: Code generation/optimization for sparse linear algebra routines

- **FFTW**: Self-adaptive DFT library + DFT kernel generator

- **SPIRAL**: Code generation/optimization for linear signal transforms

Proceedings of the IEEE special issue, Feb. 2005
a new breed of domain-aware approaches/tools push automation beyond what is currently possible applies for software and hardware design alike
SPIRAL www.spiral.net

Sponsors:
DARPA
NSF ACR-0234293
NSF ITR/NGS-0325687

and:
Cylab, CMU
Austrian Science Fund
Intel
ITRI, Taiwan
ENSCO, Inc.

~40 Publications

Overview paper:
Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo,
SPIRAL: Code Generation for DSP Transforms,
Proceedings of the IEEE

Team:
James C. Hoe (ECE, CMU)
Jeremy Johnson (CS, Drexel)
José M. F. Moura (ECE, CMU)
David Padua (CS, UIUC)
Markus Püschel (ECE, CMU)
Manuela Veloso (CS, CMU)
Robert W. Johnson (Quarry Comp. Inc.)
Christoph Überhuber (Math, TU Wien)

Students/PostDocs:
Bryan W. Singer (CS, CMU)
Jianxin Xiong (CS, UIUC)
Srinivas Chellappa (ECE, CMU)
Franz Franchetti (ECE, CMU, before TU Vienna)
Aca Gacic (ECE, CMU)
Yevgen Voronenko (ECE, CMU)
Anthony Breitzman (CS, Drexel)
Kang Chen (CS, Drexel)
Pinit Kumhom (ECE, Drexel)
Peter Tummeltshammer (CS, TU Vienna)
...
Spiral

- Code generation from scratch for linear digital signal processing (DSP) transforms (DFT, DCT, DWT, filters, ...)

- Automatic optimization and platform-tuning at the algorithmic level and the code level

- Different code types supported (scalar, vector, FMA, fixed-point, multiplierless, ...)

**Goal:** A flexible, extensible code generation framework that can survive time (to whatever extent possible) for an entire domain of algorithms

**Research question:** To what extent is it possible to abolish handcoding and handoptimization?
Code Generation and Tuning as Optimization Problem

\( T \) a DSP transform to be implemented
\( P \) the target platform
\( \mathcal{I} = \mathcal{I}(T, P) \) set of possible implementations of \( T \) on \( P \)
\( C = C(T, I, P) \) cost of implementation \( I \) of \( T \) on \( P \)

The implementation of \( T \) that is tuned to \( P \) is given by:

\[
\hat{I} = \hat{I}(P) = \arg \min_{I \in \mathcal{I}(P)} C(T, P, I)
\]

Problems:
- How to characterize and generate the set of implementations?
- How to efficiently minimize \( C \)?

Spiral exploits the domain-specific structure to implement a solver for this optimization problem
Spiral’s architecture

Domain knowledge:
Generating algorithms & manipulating algorithms

Architecture knowledge:
by evaluating runtime
From Transform to Algorithm (Formula)

Input:
Transform specification

Output:
Fast algorithm as formula

Domain Knowledge I:
Generating the algorithm space
DSP Algorithms: Example 4-point DFT

Cooley/Tukey FFT (size 4):

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

- Fourier transform
- Diagonal matrix (twiddles)
- Kronecker product
- Identity
- Permutation

- Mathematical notation exhibits structure: SPL (signal processing language)
- Suitable for computer representation
- Contains all information to generate code
SPL: Definition (BNF)

- Description language for linear DSP algorithms

**Definition (BNF):**

\[
\langle \text{spl} \rangle ::= \langle \text{generic} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{transform} \rangle \mid \\
\langle \text{spl} \rangle \cdots \langle \text{spl} \rangle \mid \\
\langle \text{spl} \rangle \oplus \cdots \oplus \langle \text{spl} \rangle \mid \\
\langle \text{spl} \rangle \otimes \cdots \otimes \langle \text{spl} \rangle \mid \\
I_n \otimes_k \langle \text{spl} \rangle \mid I_n \otimes^k \langle \text{spl} \rangle \mid \\
\langle \text{spl} \rangle
\]

- (product)
- (direct sum)
- (tensor product)
- (overlapped tensor product)
- (conversion to real)

Some Definitions:

\[
A \oplus B = \begin{bmatrix} A \\ B \end{bmatrix}
\]

\[
A \otimes B = \begin{bmatrix} a_{k,\ell} B \end{bmatrix}, \quad \text{where } A = \begin{bmatrix} a_{k,\ell} \end{bmatrix}
\]

\[
F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

\[
R_\alpha = \begin{bmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{bmatrix}
\]

\[
I_n \otimes A = \begin{bmatrix} A \\ \vdots \\ A \end{bmatrix}
\]
DSP Algorithms: Spiral Terminology

Transform

\[ DFT_n \quad \text{parameterized matrix} \]

Rule

\[ DFT_{nm} \rightarrow (DFT_n \otimes I_m) \cdot D \cdot (I_n \otimes DFT_m) \cdot P \]

- a breakdown strategy
- product of sparse matrices

Ruletree

\[ DFT_8 \]
\[ \overline{DFT_2 \quad DFT_4} \]
\[ \overline{DFT_2 \quad DFT_2} \]

- recursive application of rules
- uniquely defines an algorithm
- efficient representation
- easy manipulation

Formula

\[ DFT_8 = (F_2 \otimes I_4) \cdot D \cdot (I_2 \otimes (I_2 \otimes F_2 \cdots)) \cdot P \]

- few constructs and primitives
- uniquely defines an algorithm
- can be translated into code
Some Transforms

\[
\begin{align*}
\text{DCT-2}_n &= \left[ \cos(k(2\ell+1)\pi/2n) \right]_{0 \leq k, \ell < n}, \\
\text{DCT-3}_n &= \text{DCT-2}^T_n \quad \text{(transpose)}, \\
\text{DCT-4}_n &= \left[ \cos((2k+1)(2\ell+1)\pi/4n) \right]_{0 \leq k, \ell < n}, \\
\text{IMDCT}_n &= \left[ \cos((2k+1)(2\ell+1+n)\pi/4n) \right]_{0 \leq k < 2n, 0 \leq \ell < n}, \\
\text{RDFT}_n &= \left[ r_{k\ell} \right]_{0 \leq k, \ell < n}, \quad r_{k\ell} = \begin{cases} 
\cos \frac{2\pi k\ell}{n}, & k \leq \left\lfloor \frac{n}{2} \right\rfloor \\
-\sin \frac{2\pi k\ell}{n}, & k > \left\lfloor \frac{n}{2} \right\rfloor 
\end{cases}, \\
\text{WHT}_n &= \begin{bmatrix} \text{WHT}_{n/2} & \text{WHT}_{n/2} \\
\text{WHT}_{n/2} & -\text{WHT}_{n/2} \end{bmatrix}, \quad \text{WHT}_2 = \text{DFT}_2, \\
\text{DHT} &= \left[ \cos(2k\ell\pi/n) + \sin(2k\ell\pi/n) \right]_{0 \leq k, \ell < n}.
\end{align*}
\]

Spiral currently contains 36 transforms
Some Breakdown Rules

\[
\begin{align*}
\text{DFT}_n & \to (\text{DFT}_k \otimes I_m) \cdot T^n_m(I_k \otimes \text{DFT}_m) \cdot L^n_k, \quad n = km \\
\text{DFT}_n & \to P_n(\text{DFT}_k \otimes \text{DFT}_m)Q_n, \quad n = km, \gcd(k, m) = 1 \\
\text{DFT}_p & \to R^T_p(I_1 \oplus \text{DFT}_{p-1})D_p(I_1 \oplus \text{DFT}_{p-1})R_p, \quad p \text{ prime} \\
\text{DCT-3}_n & \to (I_m \oplus J_m) \cdot L^n_m(\text{DCT-3}_m(1/4) \oplus \text{DCT-3}_m(3/4)) \\
& \quad \cdot (F_2 \otimes I_m) \begin{bmatrix} I_m & 0 \oplus -J_{m-1} \\ \frac{1}{\sqrt{2}}(I_1 \oplus 2I_m) \end{bmatrix}, \quad n = 2m \\
\text{DCT-4}_n & \to S_n \cdot \text{DCT-2}_n \cdot \text{diag}_{0 \leq k < n}(1/(2 \cos((2k + 1)\pi/4n))) \\
\text{IMDCT}_{2m} & \to (J_m \oplus I_m \oplus I_m \oplus J_m) \left( (\begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes I_m) \oplus (\begin{bmatrix} -1 \\ -1 \end{bmatrix} \otimes I_m) \right) J_{2m} \cdot \text{DCT-4}_{2m} \\
\text{WHT}_{2^k} & \to \prod_{i=1}^{t} (I_2^{k_1+\cdots+k_{i-1}} \otimes \text{WHT}_{2^{k_i}}) \otimes I_2^{k_{i+1}+\cdots+k_t}, \quad k = k_1 + \cdots + k_t \\
\text{DFT}_2 & \to F_2 \\
\text{DCT-2}_2 & \to \text{diag}(1, 1/\sqrt{2}) F_2 \\
\text{DCT-4}_2 & \to J_2 R_{13\pi/8}
\end{align*}
\]

Base case rules

Spiral contains 100+ rules
Some Breakdown Rules for Filters

\[ \text{Filt}_n(h(z)) \rightarrow I_{[\frac{n}{b}]} \otimes _{l+r} \left( \begin{bmatrix} \frac{t+r}{b} \\ \vdots \\ 0 \end{bmatrix} T_b(h(z) z^{l-i}) \oplus k \right) T_k \left( h(z) z^{l-\left[ \frac{t+r}{b} \right] b-k} \right) \]

\[ \text{Filt}_n(h(z)) \rightarrow L_n^\frac{n}{2} \text{Filt}_n^\frac{n}{2} \left( \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \right) \cdot \left( \begin{bmatrix} h_0(z) \\ h_1(z) \\ h_0(z) + h_1(z) \end{bmatrix} \right) \cdot \text{Filt}_{n+r+l}^\frac{n+r-l}{2} \left( \begin{bmatrix} 1 & -1 \\ z & -1 \\ 0 & 1 \end{bmatrix} \right) \cdot L_2^{n+r+l} \]

\[ \text{Filt}_n(h(z)) \rightarrow R_{n, l, r}^{\text{zero}} \cdot C_{n+l+r}(h(z)) \]

\[ C_n(h(z)) \rightarrow \text{RDFT}_n^{-1} \cdot X(\hat{h}) \cdot \text{RDFT}_n, \]
\[ \hat{h} = \text{RDFT}_n \cdot h \]

Formula- (Algorithm) generation

Transform:

```
DCT-2
```

Ruletree:

```
DCT-2
    /
   /  
DCT-4  DCT-2
```

(many possibilities)

Formula:

```
L^4_2(diag(1, 1/\sqrt{2}) F_2 \oplus J_2 R_{13\pi/8})(F_2 \otimes I_2)(I_2 \oplus J_2)
```

Remaining task

```
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;
}
```

(fast)

C Code:
Set of Algorithms

- **Given a transform:**
  - Apply breakdown rules recursively until all occurring transforms are expanded
  - Choice of rules at each step yields (usually) exponentially large algorithms space:
    - about equal in operations count
    - differ in data flow

<table>
<thead>
<tr>
<th>k</th>
<th># DFTs, size $2^k$</th>
<th># DCT IV, size $2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>126</td>
</tr>
<tr>
<td>4</td>
<td>296</td>
<td>31242</td>
</tr>
<tr>
<td>5</td>
<td>27744</td>
<td>1924443362</td>
</tr>
<tr>
<td>6</td>
<td>162570361280</td>
<td>7343815121631354242</td>
</tr>
<tr>
<td>7</td>
<td>~1.01 • 10^27</td>
<td>~1.07 • 10^38</td>
</tr>
<tr>
<td>8</td>
<td>~2.31 • 10^61</td>
<td>~2.30 • 10^76</td>
</tr>
<tr>
<td>9</td>
<td>~2.86 • 10^133</td>
<td>~1.06 • 10^153</td>
</tr>
</tbody>
</table>
From Algorithm (Formula) to Optimized Algorithm

Input:
Fast algorithm as formula

Output:
Optimized formula

Domain Knowledge II: Optimizing an algorithm
Motivation: Loop Fusion

Solution: Σ-SPL and Formula manipulation

no compiler does that
Formula Level Optimization

- **Main goals:**
  - Fusing iterative steps (fusing loops), e.g., permutations with loops
  - Improving structure (data flow) for SIMD instructions

- Overcomes compiler limitations

- Formula manipulation through mathematical rules

- Implemented using multiple levels of rewriting systems

- Puts math knowledge into the system
Structure of Loop Optimization

**SPL formula**

- To $\Sigma$-SPL
  - $\Sigma$ -SPL formula
    - Join permutations
      - $\Sigma$ -SPL formula
        - Join diagonals and monomials
          - $\Sigma$ -SPL formula

$$
T_{n}^{mn}(I_{m} \otimes C_{n}) \eta_{m}^{mn}
$$

$$
\text{diag} \left( t_{n}^{mn} \right) \left( \sum_{j=0}^{n-1} S_{(j)m \otimes n} \ C_{n} \ G_{(j)m \otimes n} \right) \text{perm} \left( \epsilon_{m}^{mn} \right)
$$

$$
\text{diag} \left( t_{n}^{mn} \right) \left( \sum_{j=0}^{n-1} S_{(j)m \otimes n} \ C_{n} \ G_{n \otimes (j)m} \right)
$$

$$
\sum_{j=0}^{n-1} S_{(j)m \otimes n} \text{diag} \left( t_{n}^{mn} \circ (j)m \otimes n \right) C_{n} \ G_{n \otimes (j)m}
$$

**Rules:**

- $G_{r} \text{ perm } (\pi) = G_{\pi \circ r}$,  \( \epsilon_{m}^{mn} \circ (j)m \otimes n = \eta_{m} \otimes (j)m \)

$$
\text{diag } (f) \ S_{w} = S_{w} \text{ diag } (f \circ w)
$$
Loop Fusion Beyond Cooley-Tukey

Main DFT recursion (breakdown rules):

\[ \text{DFT}_{km} \rightarrow (\text{DFT}_k \otimes I_m) \mathcal{T}^n_m (I_k \otimes \text{DFT}_m) \mathcal{L}^n_k \]

\[ \text{DFT}_{km} \rightarrow P_n (\text{DFT}_k \otimes \text{DFT}_m) Q_n, \quad \gcd(k, m) = 1 \]

\[ \text{DFT}_p \rightarrow R_p^T (I_1 \oplus \text{DFT}_{p-1}) D_p (I_1 \oplus \text{DFT}_{p-1}) R_p \]

\[ \text{DFT}_n \rightarrow D_m \text{DFT}_m D'_m \text{DFT}_m D''_m, \quad m > 2n \]

How to fuse permutations from different combinations of rules?
Example

- Consider the DFT formula fragment

\[
(I_p \otimes (I_1 \oplus (I_r \otimes \text{DFT}_s) L_r^S ) W_p ) V_{p,q}
\]

- In Σ-SPL:

\[
\sum_{j_1=0}^{p-1} \left( S((j_1)_p \otimes \iota_q) \circ (0)^{1 \rightarrow q} \circ \iota_1 \ G_{v,p,q} \circ ((j_1)_p \otimes \iota_q) \circ \bar{w}_{1,g} \circ (0)^{1 \rightarrow q} \right) \\
+ \sum_{j_0=0}^{r-1} S((j_1)_p \otimes \iota_q) \circ (1)^{q-1 \rightarrow q} \circ \iota_0 \circ ((j_0)_r \otimes \iota_s) \ \text{DFT}_s
\]

- Complicated array access

\[
G_{v,p,q} \circ ((j_1)_p \otimes \iota_q) \circ \bar{w}_{1,g} \circ (1)^{q-1 \rightarrow q} \circ \ell_r^S \circ ((j_0)_r \otimes \iota_s)
\]
Example (cont’d)

After index function simplification:

\[
\sum_{j_1=0}^{p-1} \left( S_{h_{0,q} \to p,q \circ (j_1)_p} G_{\tilde{h}_{0,q} \to p,q \circ (j_1)_p} + \sum_{j_0=0}^{r-1} S_{h_{q,j_1+s,j_0+1,1}} \text{DFT}_s G_{\tilde{h}_{b_1,p \to q \circ w^{s \to q}_{\phi_1,g^s}}} \right)
\]

Simplified array access
Example (cont’d)

- **Generated Code**

```c
// Input: _Complex double x[28], output: y[28]
int p1, b1;
for(int j1 = 0; j1 <= 3; j1++) {
    y[7*j1] = x[(7*j1)%28];
    p1 = 1; b1 = 7*j1;
    for(int j0 = 0; j0 <= 2; j0++) {
        y[b1 + 2*j0 + 1] =
        x[(b1 + 4*p1)%28] + x[(b1 + 24*p1)%28];
        y[b1 + 2*j0 + 2] =
        x[(b1 + 4*p1)%28] - x[(b1 + 24*p1)%28];
        p1 = (p1*3%7);
    }
}
```
Vector code generation from SPL formulas

Naturally vectorizable construct

\[ A \otimes I_4 \]

vector length

(Current) generic construct completely vectorizable:

\[
\prod_{i=1}^{k} P_i D_i (A_i \otimes I_\nu) E_i Q_i
\]

\[ P_p, Q_i \] permutations
\[ D_p, E_i \] diagonals
\[ A_i \] arbitrary formulas
\[ \nu \] SIMD vector length

Vectorization in two steps:

1. Formula manipulation using manipulation rules
2. Code generation (vector code + C code)

Formula manipulation overcomes compiler limitations
Example DFT

Standard FFT

\[(\text{DFT}_k \otimes I_m)^T (I_k \otimes \text{DFT}_m)^T\]

Formula manipulation

\[\left( I_{mn \nu} \otimes L^{2\nu}_{\nu} \right) \left( \text{DFT}_m \otimes I_{n \nu} \otimes I_{\nu} \right) \left( L_{mn \nu} \otimes L^{2\nu}_{2\nu} \right)\]

Vector FFT for \(\nu\)-way vector instructions
Implementation of Formula Generation and Manipulation

- Implementation using a computer algebra system (GAP)
- SPL/Σ-SPL implemented as recursive data types
- Exact representation of sin(), cos(), etc.
- Symbolic computation enables exact verification of rules
From Optimized Algorithm (Formula) to Code

Input:
Optimized formula

Output:
Intermediate Code

Straightforward
From Code to Optimized Code

Input: Intermediate Code

Output: Optimized C code
Code Level Optimizations

- Precomputation of constants
- Loop unrolling (controlled by search module)
- Constant inlining
- SSA code, scalar replacement, algebraic simplifications, CSE
- Code reordering for locality (optional)
- Conversion to FMA code (optional)
- Conversion to fixed point code (optional)
- Conversion to multiplierless code (optional)

- Finally: Unparsing to C (or Fortran)
Conversion to FMA code

- FMA (fused multiply-add) or MAC (multiply accumulate) instructions: $y = \pm ax \pm b$

- Extension of the instruction set + specialized execution units

- As fast as a single add or multiply

- Conversion of linear algorithms to FMA code: blackboard

- Paper: Yevgen Voronenko and Markus Püschel
  [Automatic Generation of Implementations for DSP Transforms on Fused Multiply-Add Architectures](http://example.com)
  Proc. (ICASSP) 2004
Evaluating Code

Input:
- Optimized C code

Output:
- Performance Number

Straightforward

Examples:
- runtime
- accuracy
- operations count
Search (Learning) for the Best

Input:
Performance Number

Output:
Controls Formula Generation
Controls Implementation Options
Search Methods

- **Search over:**
  - Algorithmic degrees of freedom (choice of breakdown rules)
  - Implementation degrees of freedom (degree of unrolling)

- **Operates with the ruletree representation of an algorithm**
  - transform independent
  - efficient

- **Search Methods**
  - Exhaustive Search
  - Dynamic Programming (DP)
  - Random Search
  - Hill Climbing
  - STEER (an evolutionary algorithm)
STEER: Evolutionary Search

Population n:

......

......

Population n+1:

Mutation

Cross-Breeding

Survival of Fittest

expand differently

swap expansions
Learning

Procedure:
- Generate a set of (1000 say) algorithms and their runtimes (one transform, one size); represent algorithms by features
- From this data (pairs of features and runtimes), learn a set of algorithm design rules
- From this set, generate best algorithms (theory of Markov decision processes)

Evaluation:
- Tested for WHT and DFT
- From data generated for one size ($2^{15}$) could construct best algorithms across sizes ($2^{12}$-$2^{18}$)

Bryan Singer and Manuela Veloso
Learning to Construct Fast Signal Processing Implementations
Benchmarks
Benchmark: DFT, 2-powers

P4, 3.2 GHz, icc 8.0

Vendor library: handtuned assembly?

Higher is better

Single precision
Benchmark: DFT, Other Sizes

- Divide sizes into levels by number of necessary Rader steps
- \( n < 8192 \)
Benchmark: DFT, Level 1 Sizes

Level 1 Sizes

- FFTW 3.0.1
- FFTW 3.0.1 SSE2
- SPIRAL Rader
- SPIRAL Rader SSE3
- SPIRAL Bluestein SSE3

Runtime [s]

DFT Size

2-4 x
Benchmark: DFT, Level 2 Sizes

- FFTW 3.0.1
- FFTW 3.0.1 SSE2
- SPIRAL Rader
- SPIRAL Rader SSE3
- SPIRAL Bluestein SSE3

Runtime [s]

DFT Size

Level 2 Sizes

2-5 x
Benchmark: DFT, Level 3 Sizes

Level 3 Sizes

Run time [s]

DFT Size

FFT 3.0.1
FFT 3.0.1 SSE2
SPIRAL Rader
SPIRAL Rader SSE3
SPIRAL Bluestein SSE3

4-9 x
Benchmark: Fixed Point DFT, IPAQ

IPAQ
Xscale arch.
400 MHz
Has only fixed point

Higher is better

Intel spent less effort?
Benchmark: DCT

P4, 3.2 GHz, icc 8.0

- This is not the latest IPP
- Spiral gains a factor of 2 to vendor library
- Another factor of 3 with 2D and vector instructions
Benchmark: Filter (Relative to IPP)

Lower = better
Instructive Experiments
Performance Spread: DCT, size 32
Histograms, 10,000 algorithms

runtime: x2
#ops: x1.08
#assembly instr: x1.5

#ops vs. runtime: no correlation
#fma ops: x1.2
accuracy: x10, most x2

P4, 3.2 GHz, gcc
Performance Spread: DFT $2^{16}$
Histograms, 20,000 Algorithms

- Generality of vectorization (all algorithms improve)
- Efficiency of vectorization (x 2.5 gain)
Performance Spread: Filter(128, 16)

Pentium 4 – 3.2
Filter: Time Domain Methods

\[ \text{Filt}_n(h(z)) \rightarrow I_n \otimes_{k-1} (h_0, \ldots, h_{k-1}) \]

for \( i = 0 \ldots n-1 \)
\[ y[i] = h[0]*x[0+i]+h[1]*x[1+i]+\ldots+h[n-1]*x[n-1+i] \]
end

Xeon-1.7

![Graph showing relative run times vs. base for different blocking strategies. The graph has a logarithmic x-axis labeled "size (log_2 n)" and a linear y-axis labeled "relative run times vs. base." The best blocking strategy is indicated.]
Filter: All Methods

Athlon XP 1.73

- 16: Time domain wins
- 32: Karatsuba wins
- 64: Karatsuba/DFT ~equal
Platform Dependency: DFT

50% Loss by porting from PIII to P4
## Platform Dependency: Filter

<table>
<thead>
<tr>
<th></th>
<th>16-tap</th>
<th>32-tap</th>
<th>64-tap</th>
<th>128-tap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4 3.0GHz</td>
<td>Blocking</td>
<td>Karatsuba</td>
<td>RDFT</td>
<td>RDFT</td>
</tr>
<tr>
<td>Northwood</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4 3.6GHz</td>
<td>Blocking</td>
<td>Karatsuba</td>
<td>Karatsuba</td>
<td>RDFT</td>
</tr>
<tr>
<td>Prescott</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macintosh</td>
<td>Karatsuba</td>
<td>Karatsuba</td>
<td>RDFT</td>
<td>RDFT</td>
</tr>
<tr>
<td>Xeon 1.7 GHz</td>
<td>Blocking</td>
<td>Blocking</td>
<td>Blocking</td>
<td>RDFT</td>
</tr>
<tr>
<td>Athlon 1.73GHz</td>
<td>Karatsuba/</td>
<td>Karatsuba</td>
<td>Karatsuba/</td>
<td>RDFT</td>
</tr>
<tr>
<td></td>
<td>Blocking</td>
<td></td>
<td>RDFT</td>
<td></td>
</tr>
</tbody>
</table>
Compileroptions: Filter

Macintosh - GNU C 3.3 (Apple)

Blocking/nesting  + Karatsuba

cgc {-01/-03} -fomit-frame-pointer -std=c99 -fast -mcpu=7450
Compileroptions DCT

ACOVEA: Evolutionary search for best compiler flags (gcc has ~500)

P4, 3.2 GHz, gcc

10% improvement of best Spiral generated code
Multiplierless DFT, IPAQ

- IPAQ
- Xscale arch.
- 400 MHz
- Fixed-point only

- fixed-point constant multiplications replaced by adds and shifts
- trade-off runtime and precision
Summary

- Code generation and tuning as optimization problem over the algorithm and implementation space
  
  *Exploit the structure of the domain to solve it*

- Declarative framework for computer representation of the domain-knowledge
  
  *Enables algorithm generation and optimization (teaches the system the math; does not become obsolete?)*

- Compiler to translate math description into code

- Search and learning to explore implementation space
  
  *Closes the feedback loop gives the system “intelligence”*

- Extensible, versatile
  
  *Every step in the code generation is exposed*

www.spiral.net