Short Vector SIMD Code Generation for DSP Algorithms

Franz Franchetti
Christoph Ueberhuber
Applied and Numerical Mathematics
Technical University of Vienna
Austria

Markus Püschel
José Moura
Electrical and Computer Engineering
Carnegie Mellon University

http://www.ece.cmu.edu/~spiral
http://www.math.tuwien.ac.at/~aurora
Outline

- Short vector extensions
- Digital signal processing (DSP) transforms
- SPIRAL
- Vectorization of SPL formulas
- Experimental results
SIMD Short Vector Extensions

- Extension to instruction set architecture
- Available on most current architectures
- Originally for multimedia (like MMX for integers)
- Requires fine grain parallelism
- Large potential speed-up

<table>
<thead>
<tr>
<th>Name</th>
<th>n-way</th>
<th>Precision</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>4-way</td>
<td>float</td>
<td>Intel Pentium III and 4, AMD AthlonXP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2-way</td>
<td>double</td>
<td>Intel Pentium 4</td>
</tr>
<tr>
<td>3DNow!</td>
<td>2-way</td>
<td>float</td>
<td>AMD K6, K7, AthlonXP</td>
</tr>
<tr>
<td>AltiVec</td>
<td>4-way</td>
<td>float</td>
<td>Motorola G4</td>
</tr>
<tr>
<td>IPF</td>
<td>2-way</td>
<td>Float</td>
<td>Intel Itanium, Itanium 2</td>
</tr>
</tbody>
</table>
Problems

- SIMD instructions are architecture specific
- No common API (usually assembly hand coding)
- Performance very sensitive to memory access
- Automatic vectorization (by compilers) very limited

 Requires expert programmers

Our Goal: Automation for digital signal processing (DSP) transforms
DSP (digital signal processing) transforms

sampled signal (a vector)

transform (a matrix)

\[ x \mapsto Mx \]

Example: Discrete Fourier Transform (DFT) size 4

\[
DFT_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -i & -1 & i \\
1 & -1 & 1 & -1 \\
1 & i & -1 & i \\
\end{bmatrix}
= \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & -1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & -1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\]

\[
DFT_4 = (DFT_2 \otimes I_2)D(I_2 \otimes DFT_2)P
\]

- Fast algorithm = product of structured sparse matrices
- Represented as formula using few constructs (e.g., \( \otimes \)) and primitives (diagonal, permutation)
- Captures a large class of transforms (DFT, DCT, wavelets, ...)
**Tensor (Kronecker) Product of Matrices**

\[ A \otimes B = [a_{kl} B]_{k,l} \]

for \( A = [a_{kl}]_{k,l} \)

**Examples:**

\[
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
\end{bmatrix} \otimes I_2 = \\
\begin{bmatrix}
1 & 2 & 1 & 2 \\
3 & 4 & 3 & 4 \\
\end{bmatrix}
\]

\[
I_2 \otimes \begin{bmatrix}
1 & 2 \\
3 & 4 \\
\end{bmatrix} = \\
\begin{bmatrix}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{bmatrix}
\]

**Key construct in many DSP transform algorithms (DFT, WHT, all multidimensional)**
SPIRAL: A Library Generator for Platform-Adapted DSP Transform

www.ece.cmu.edu/~spiral

Observation:

• For a given transform there are maaaany different algorithms (equal in arithmetic cost, differ in data flow)
• The best algorithm and its implementation is platform-dependent
• It is not clear what the best algorithm/implementation is

SPIRAL:

Automatic algorithm generation
+ Automatic translation into code
+ Intelligent search for “best”

= generated platform-adapted implementation
SPIRAL’s Mathematical Framework

**Transform**

\[ DFT_n \]  
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

**Formula**

\[ DFT_{16} = (DFT_4 \otimes I_4) \cdot T_4^{16} \cdot (I_4 \otimes DFT_4) \cdot L_4^{16} \]

- by recursive application of rules
- few constructs and primitives
- can be translated into code

Used as mathematical high-level representation of algorithms
(SPL = signal processing language)
SPIRAL system

DSP transform (user specified)

Formula Generator
- fast algorithm as SPL formula
- controls algorithm generation

SPL Compiler
- platform-adapted implementation
- C/Fortran/SIMD code
- runtime on given platform
- controls implementation options

Search Engine
- controls

Our Goal: extend SPL compiler to generate vector code
Generating SIMD Code from SPL Formulas

Example:

\[ y := (A \otimes I_4)x \]

naturally represents vector operation

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

- Formulas contain all structural information for vectorization
- Construct above captures DFT, WHT, all multi-dimensional

\[ P_i, Q_i \quad \text{permutations} \]
\[ D_i, E_i \quad \text{diagonals} \]
\[ A_i \quad \text{arbitrary formulas} \]
\[ \nu \quad \text{SIMD vector length} \]
The Approach

- Use macro layer as API to hide machine specifics
- Vector code generation in two steps
  1. Symbolic vectorization (formula manipulation)
  2. Code generation
Symbolic Vectorization

\[ DFT_{16} = (DFT_4 \otimes I_4) \cdot T_{16}^4 \cdot (I_4 \otimes DFT_4) \cdot L_{16}^4 \]

Formula manipulation (automatic using manipulation rules)

\[ \overline{DFT}_{16} = \left( (I_4 \otimes L_8^4) \cdot (DFT_4 \otimes I_4) \cdot \overline{T}_{16}^4 \right) \cdot \left( (I_4 \otimes L_2^8) (L_{16}^1 \otimes I_2) (I_4 \otimes L_4^8) \cdot (DFT_4 \otimes I_4) \cdot (I_4 \otimes L_2^8) \right) \]

Pattern matching

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

- Manipulate to match vectorizable construct
- Separate vectorizable parts and scalar parts
Normalizing formulas

\[(I_n \otimes L_{v}^{2v})(I_{n} \otimes L_{n}^{2v}) = I_{2nv}\]
\[A \otimes B = (A \otimes I_m)(I_n \otimes B)\]
\[I_{v} \otimes A = L_{v}^{nv}(A \otimes I_{v})L_{n}^{nv}\]
\[I_{nv+l} = I_{nv} \oplus I_{l}\]
\[I_{mn} = I_{m} \otimes I_{n}\]
\[PD = D'P\]

Converting complex to real arithmetic

\[\overline{AB} = \overline{A} \cdot \overline{B}\]
\[\overline{A} = A \otimes I_2, \quad A \text{ real}\]
\[\overline{D} = (I_{n/v} \otimes L_{v}^{2v})\overline{D'}(I_{n/v} \otimes L_{2}^{2v}), \quad V|n\]
\[A \otimes I_{v} = (I_n \otimes L_{v}^{2v})(\overline{A} \otimes I_{v})(I_n \otimes L_{2}^{2v})\]
Vector Code Generation

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

- fuse with load/store operations
- difficult part (easy to lose performance)
- arithmetic vector instructions
  - use standard SPL compiler on \(A_i\)
  - replace scalar with vector instructions
- easy part (due to existing SPL compiler)

---

\| \hline
| \(P_i, Q_i\) | permutations |
| \(D_i, E_i\) | diagonals |
| \(A_i\) | arbitrary formulas |
| \(\nu\) | SIMD vector length |
\| \hline

---
**Challenge:** Data Access

**Example:**
- **Required:**
  - Memory
  - Registers
  - Strided load of complex numbers

- **Available:**
  - Memory
  - Registers
  - Vector load plus in-register permutations

- highest performance code requires **properly aligned** data access
- permutation support differs between architectures
- performance differs between permutations (some are good, most very bad)

**Solution:**
- use formula manipulation to get “good” permutations
- macro layer API for efficient and machine transparent implementation
Portable High-level API

- restricted set of short vector operations
- requires C compiler with „intrinsics“-interface
- high-level operations
  - Vector arithmetic operations
  - Vector load/store operations
  - Special and arbitrary multi-vector permutations
  - Vector constant handling (declaration, usage)
  - Implemented by C macros

Example:

Unit-stride load of 4 complex numbers:

```
LOAD_L_8_2(reg1, reg2, *mem)
```
Portable SIMD API: Details

All SIMD extensions supported:

- gcc 3.0, gcc-vec
- Intel C++ Compiler, MS VisualC++ with ProcessorPack
- Various PowerPC compilers (Motorola standard)

Examples:

Reverse load of 4 real numbers:

\[ \text{LOAD}_J\_4(\text{reg, *mem}) \]

Reverse load of 4 complex numbers:

\[ \text{LOAD}_J\_4\_x\_I\_2(\text{r1, r2, *mem}) \]
Generated Code

- Vector parts: portable SIMD API
- Scalar parts: standard C
- $P_i, Q_i, D_i, E_i$ handled by load/store operations
- $A_i$ handled by vector arithmetics

```c
void DFT_16(vector_float *y, vector_float *x)
{
    vector_float x10, x11, x12;
    ...
    LOAD_VECT(x10, x + 0);
    LOAD_VECT(x14, x + 16);
    f0 = SIMD_SUB(x10, x14);
    LOAD_VECT(x11, x + 4);
    LOAD_VECT(x15, x + 20);
    f1 = SIMD_SUB(x11, x15);
    ...
    yl7 = SIMD_SUB(f1, f4);
    STORE_L_8_4(yl6, yl7, y + 24);
    yl2 = SIMD_SUB(f0, f5);
    yl3 = SIMD_ADD(f1, f4);
    STORE_L_8_4(yl2, yl3, y + 8);
}
```

```c
/*  Intel SSE: portable SIMD API */
/*  Intel C++ Compiler 5.0 */
typedef __m128 vector_float;

#define LOAD_VECT(a, b)                     
    (a) = *(b)

#define SIMD_ADD(a, b)                     
    _mm_add_ps((a), (b))

#define SIMD_SUB(a, b)                     
    _mm_sub_ps((a), (b))

#define STORE_L_8_4(re, im, out)           
    {
        vector_float _sttmp1, _sttmp2;
        _sttmp1 = _mm_unpacklo_ps(re, im);  
        _sttmp2 = _mm_unpackhi_ps(re, im);  
        _mm_store_ps(out, _sttmp1);       
        _mm_store_ps((out) + VLEN, _sttmp2);
    }
```
Experimental Results

- our code is generated, found by dynamic programming search
- different searches for different types of code (scalar, vector)
- results in (Pseudo) gigaflops (higher = better)
DFT code: Pentium 4, SSE

(Pseudo) gflops

DFT $2^n$ single precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0

speedups (to C code) up to factor of 3.1

hand-tuned vendor assembly code
Generated DFT Code: Pentium 4, SSE2

DFT $2^n$ double precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0

speedups (to C code) up to factor of 1.8
Generated DFT Code: Pentium III, SSE

DFT $2^n$ single precision, Pentium III, 1 GHz, using Intel C compiler 6.0

speedups (to C code) up to factor of 2.1
DFT Code: Athlon XP, SSE

DFT $2^n$ single precision, Pentium III, 1 GHz, using Intel C compiler 6.0

speedups (to C code) up to factor of 1.6
Other transforms

- WHT has only additions
- very simple transform

speedups (to C code) up to factor of 3

WHT $2^n$
Pentium 4, 2.53 GHz, SSE

2-dim DCT $2^n \times 2^n$
Pentium 4, 2.53 GHz, SSE
Different search strategies

DFT $2^n$ single precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0

standard DP looses up to 25% performance
### Best DFT Trees, size $2^{10} = 1024$

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4 float</th>
<th>Pentium 4 double</th>
<th>Pentium III float</th>
<th>AthlonXP float</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td><img src="image1" alt="Tree" /></td>
<td><img src="image2" alt="Tree" /></td>
<td><img src="image3" alt="Tree" /></td>
<td><img src="image4" alt="Tree" /></td>
</tr>
<tr>
<td>C vect</td>
<td><img src="image5" alt="Tree" /></td>
<td><img src="image6" alt="Tree" /></td>
<td><img src="image7" alt="Tree" /></td>
<td><img src="image8" alt="Tree" /></td>
</tr>
<tr>
<td>SIMD</td>
<td><img src="image9" alt="Tree" /></td>
<td><img src="image10" alt="Tree" /></td>
<td><img src="image11" alt="Tree" /></td>
<td><img src="image12" alt="Tree" /></td>
</tr>
</tbody>
</table>

platform/datatype dependent
Crosstiming of best trees on Pentium 4

DFT $2^n$ single precision, runtime of best found of other platforms

binary compatibility is not performance compatibility
Summary

- Automatically generated vectorized DSP code
- Code platform-adapted (SPIRAL)
- We implement “constructs”, not transforms
- Very competitive performance
- DFT, WHT, arbitrary multi-dim supported

Ongoing work:

- port to other SIMD architectures
- include filters and wavelets

http://www.ece.cmu.edu/~spiral
http://www.math.tuwien.ac.at/~aurora