

Short Vector SIMD Code Generation for DSP Algorithms

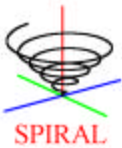
Franz Franchetti
Christoph Ueberhuber

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>



AURORA

Sponsors

Work supported by DARPA (DSO), Applied & Computational Mathematics Program, OPAL, through grant managed by research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.



AURORA

Outline

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



—AURORA—

SIMD Short Vector Extensions

vector length = 4
(4-way)



- 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**

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



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

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

$$DFT_4 = (DFT_2 \otimes I_2) D (I_2 \otimes DFT_2) P$$

- 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

$$\begin{array}{c} \text{coarse} \\ \text{structure} \end{array} A \otimes \begin{array}{c} \text{fine} \\ \text{structure} \end{array} B = [a_{kl} B]_{k,l} \quad \text{for } A = [a_{kl}]_{k,l}$$

Examples:

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

**identity
matrix**

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



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

José Moura (CMU)
Jeremy Johnson (Drexel)
Robert Johnson (MathStar)
David Padua (UIUC)
Markus Püschel (CMU)
Viktor Prasanna (USC)
Manuela Veloso (CMU)

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



AURORA

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

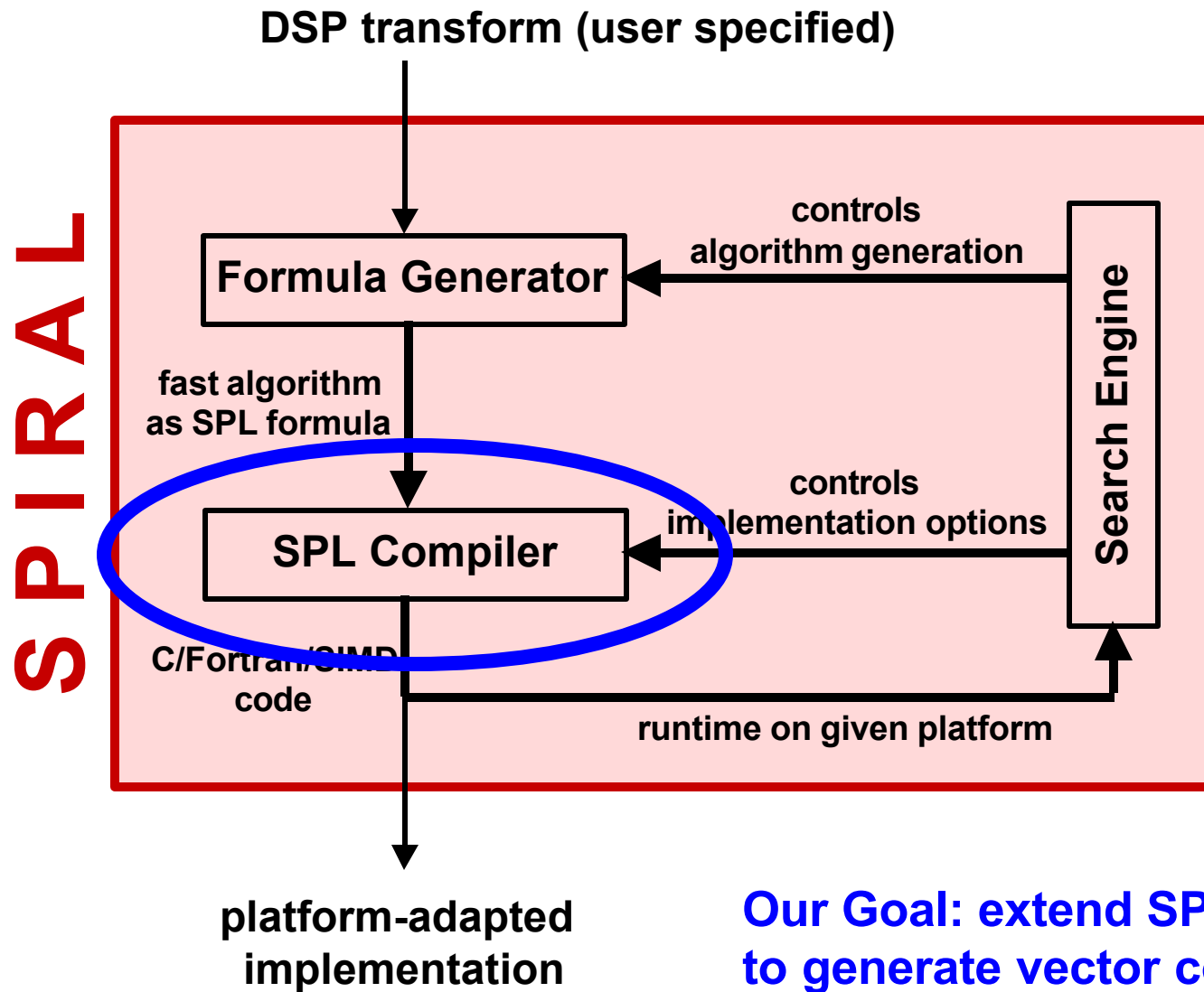
Formula $DFT_{16} = (DFT_4 \otimes I_4) \cdot T_4^{16} \cdot (I_4 \otimes DFT_4) \cdot L_4^{16}$

- 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



Our Goal: extend SPL compiler to generate vector code



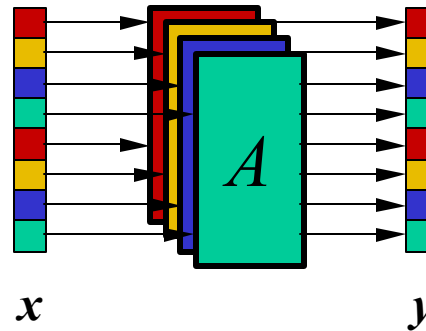
AURORA

Generating SIMD Code from SPL Formulas

Example:

$$y := (A \otimes I_4)x$$

vector length



naturally represents
vector operation

(Current) generic construct **completely vectorizable**:

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

P_i, Q_i permutations
 D_i, E_i diagonals
 A_i arbitrary formulas
 v SIMD vector length

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



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

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



Formula manipulation
(automatic using manipulation rules)

$$\overline{DFT}_{16} = \left((I_4 \otimes L_4^8) \cdot (\overline{DFT}_4 \otimes I_4) \cdot \overline{T}_4^{16} \right)$$

$$\left((I_4 \otimes L_2^8) (L_4^{16} \otimes I_2) (I_4 \otimes L_4^8) \cdot (\overline{DFT}_4 \otimes I_4) \cdot (I_4 \otimes L_2^8) \right)$$



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



AURORA

Formula Manipulation

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

$$\overline{A \cdot B} = \overline{A} \cdot \overline{B}$$

$$\overline{A} = A \otimes I_2, \quad A \text{ real}$$

$$\overline{D} = (I_{n/v} \otimes L_v^{2v}) \overline{D'} (I_{n/v} \otimes L_2^{2v}), \quad v | n$$

$$\overline{A \otimes I_v} = (I_n \otimes L_v^{2v}) (\overline{A} \otimes I_v) (I_n \otimes L_2^{2v})$$



Vector Code Generation

fuse with load/store operations

difficult part
(easy to loose performance)

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

arithmetic vector instructions

- use standard SPL compiler on A_i
- replace scalar with vector instructions

P_i	Q_i	permutations
D_i	E_i	diagonals
A_i		arbitrary formulas
$\hat{v}$		SIMD vector length

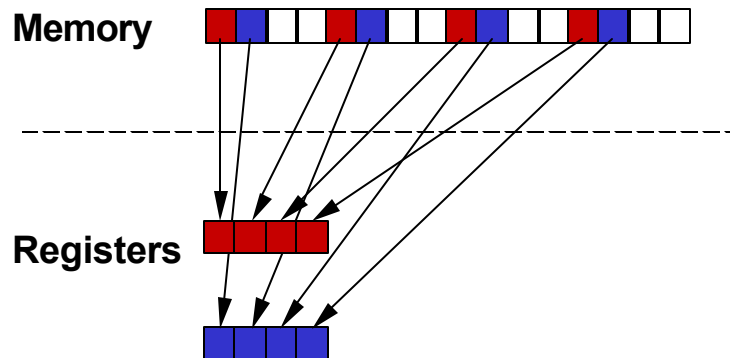
easy part
(due to existing SPL compiler)



Challenge: Data Access

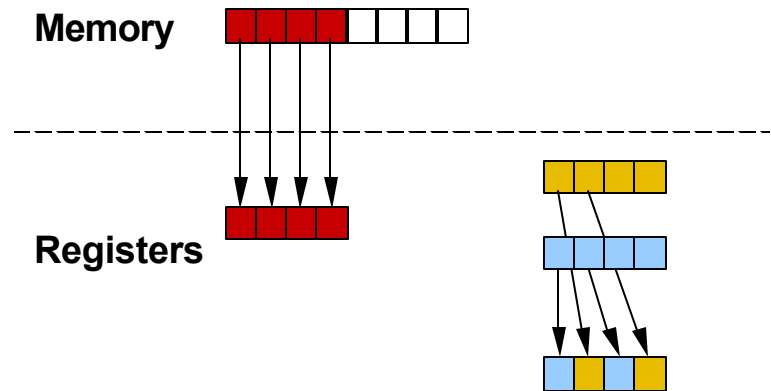
Example:

Required:



Strided load of complex numbers

Available:



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



AURORA

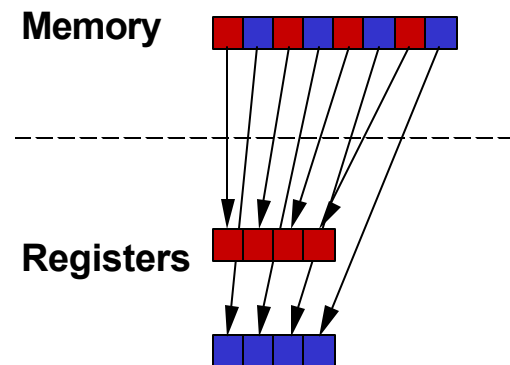
Portable High-level API

- 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

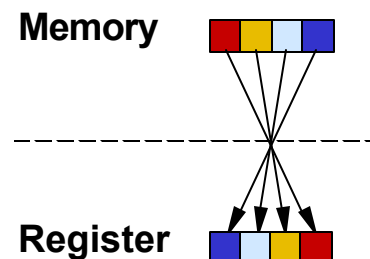
All SIMD extensions supported:

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

Examples:

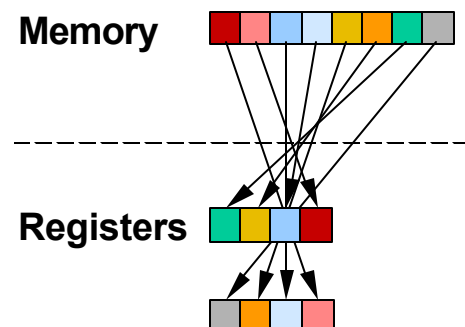
Reverse load of 4 real numbers:

```
LOAD_J_4 (reg, *mem)
```



Reverse load of 4 complex numbers:

```
LOAD_J_4_x_I_2 (r1, r2, *mem)
```



AURORA

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

```
/*    Example vector code: DFT_16    */
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);
    ...
    y17 = SIMD_SUB(f1, f4);
    STORE_L_8_4(y16, y17, y + 24);
    y12 = SIMD_SUB(f0, f5);
    y13 = SIMD_ADD(f1, f4);
    STORE_L_8_4(y12, y13, y + 8);
}
```

```
/*    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); \
}
```



AURORA

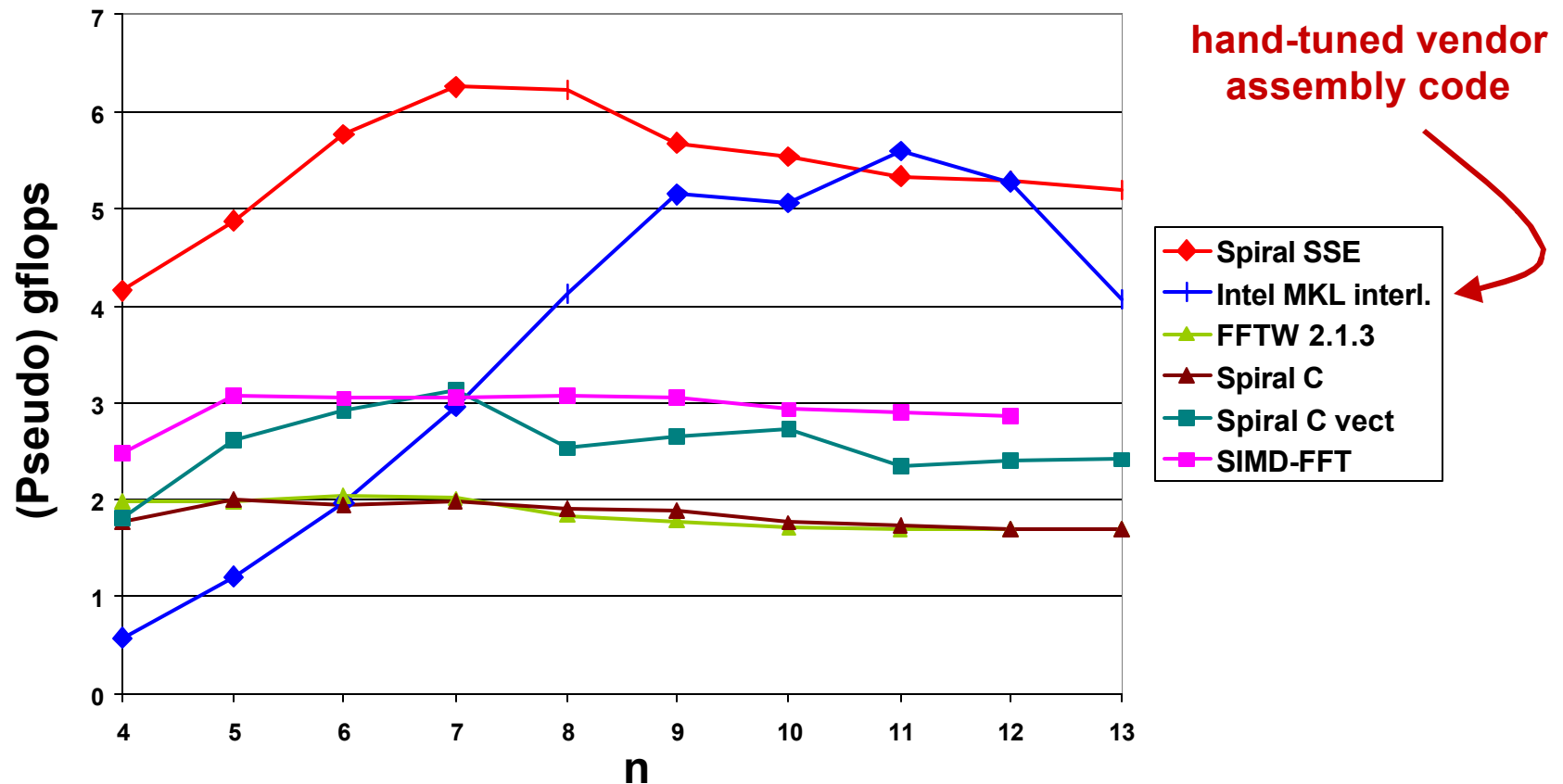
Experimental Results

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



AURORA

Generated DFT Code: Pentium 4, SSE



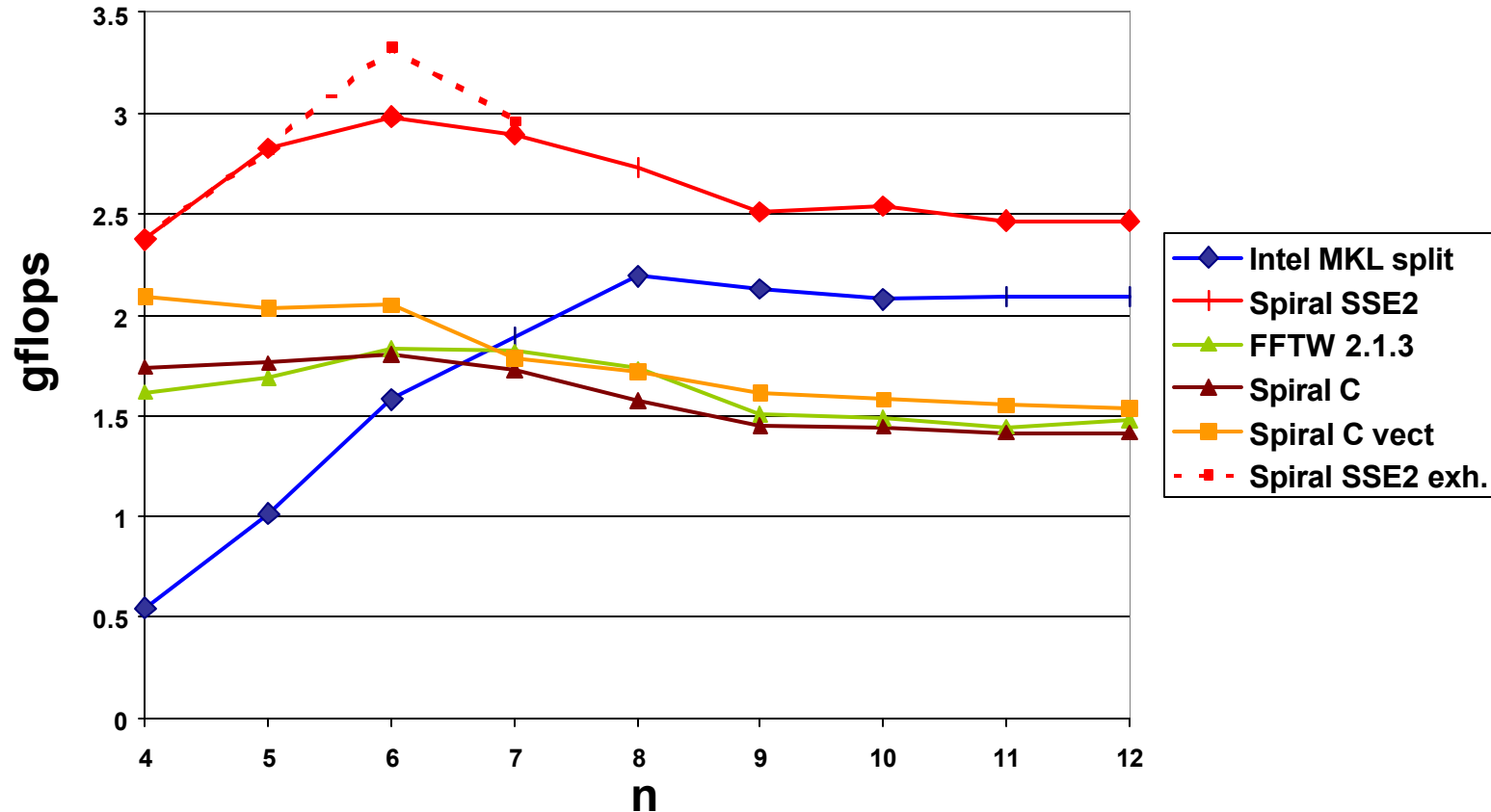
DFT 2^n single precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0



speedups (to C code) up to factor of 3.1

AURORA

Generated DFT Code: Pentium 4, SSE2



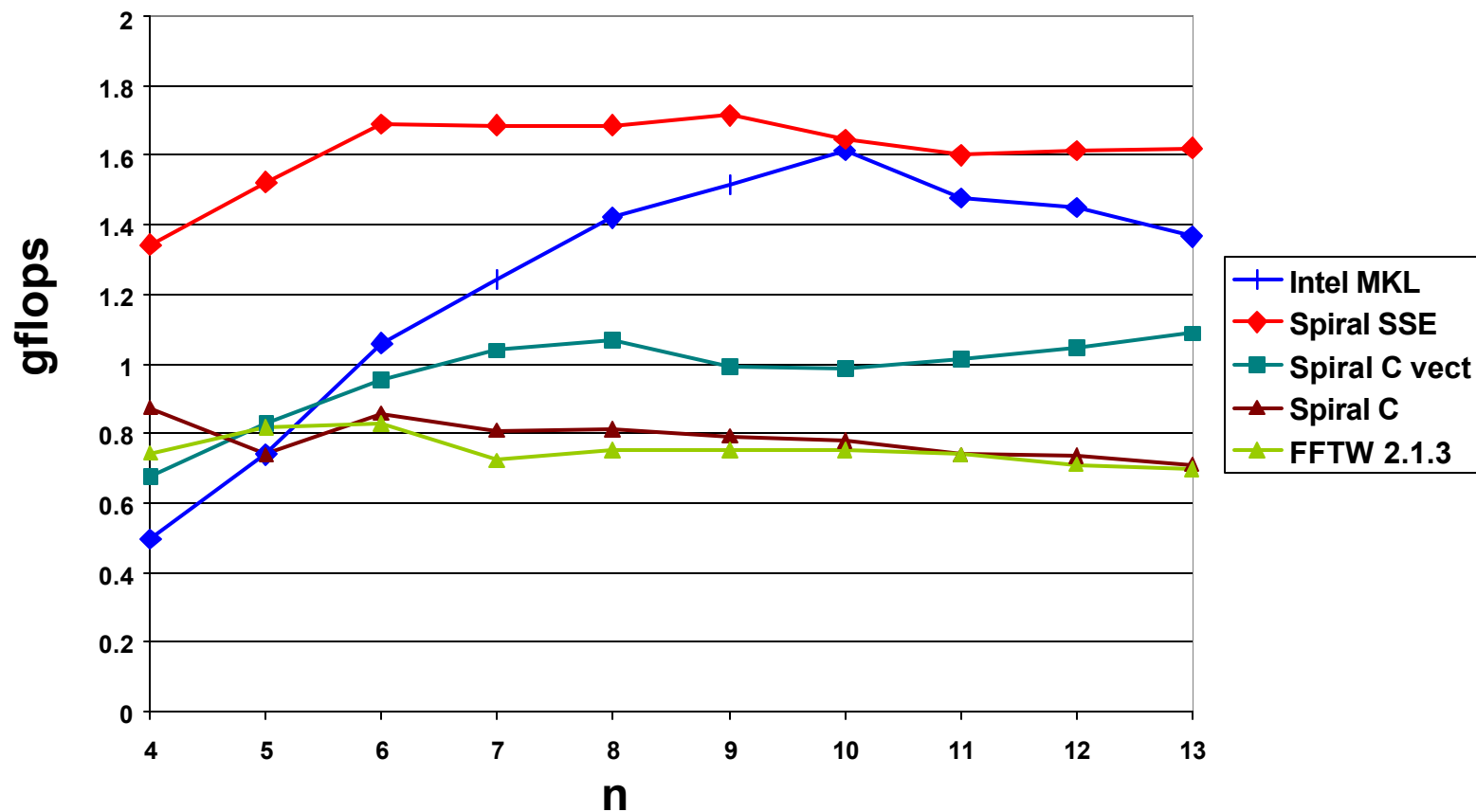
DFT 2^n *double* precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0



speedups (to C code) up to factor of 1.8



Generated DFT Code: Pentium III, SSE



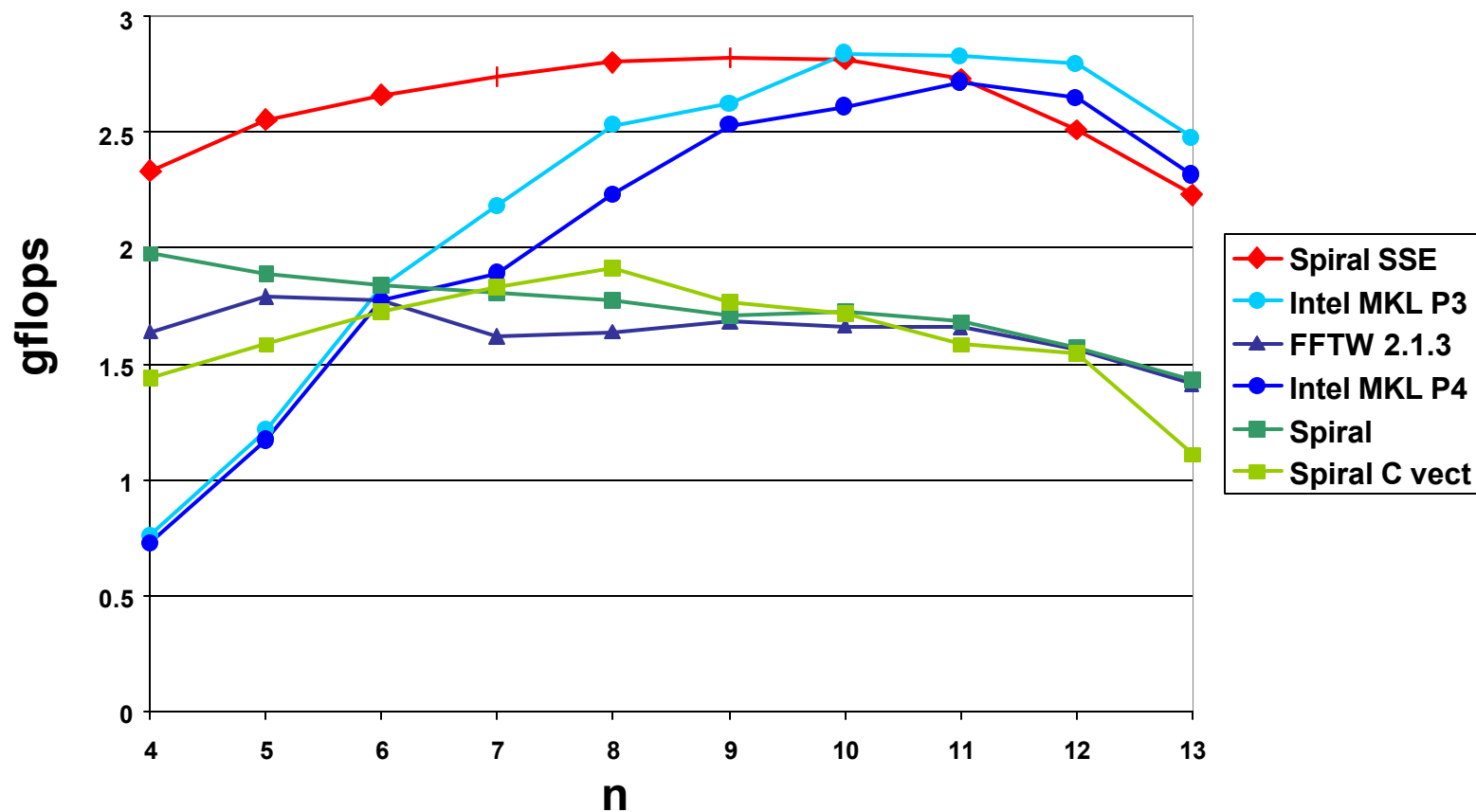
DFT 2^n single precision, Pentium III, 1 GHz, using Intel C compiler 6.0



speedups (to C code) up to factor of 2.1

AURORA

Generated DFT Code: Athlon XP, SSE



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



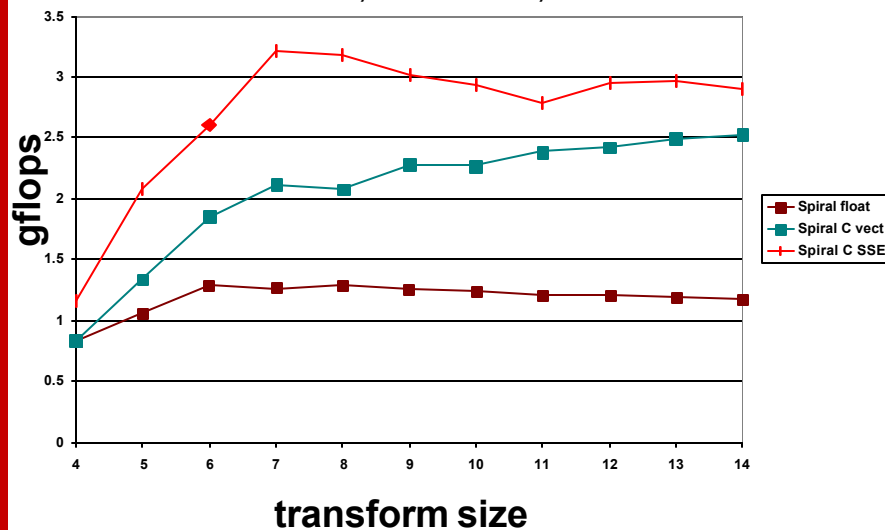
speedups (to C code) up to factor of 1.6

AURORA

Other transforms

WHT 2^n

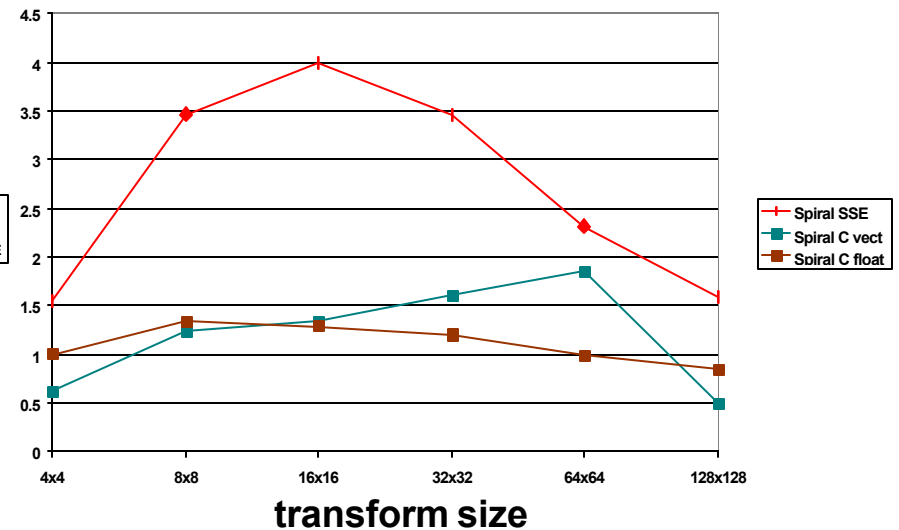
Pentium 4, 2.53 GHz, SSE



- WHT has only additions
- very simple transform

2-dim DCT $2^n \times 2^n$

Pentium 4, 2.53 GHz, SSE

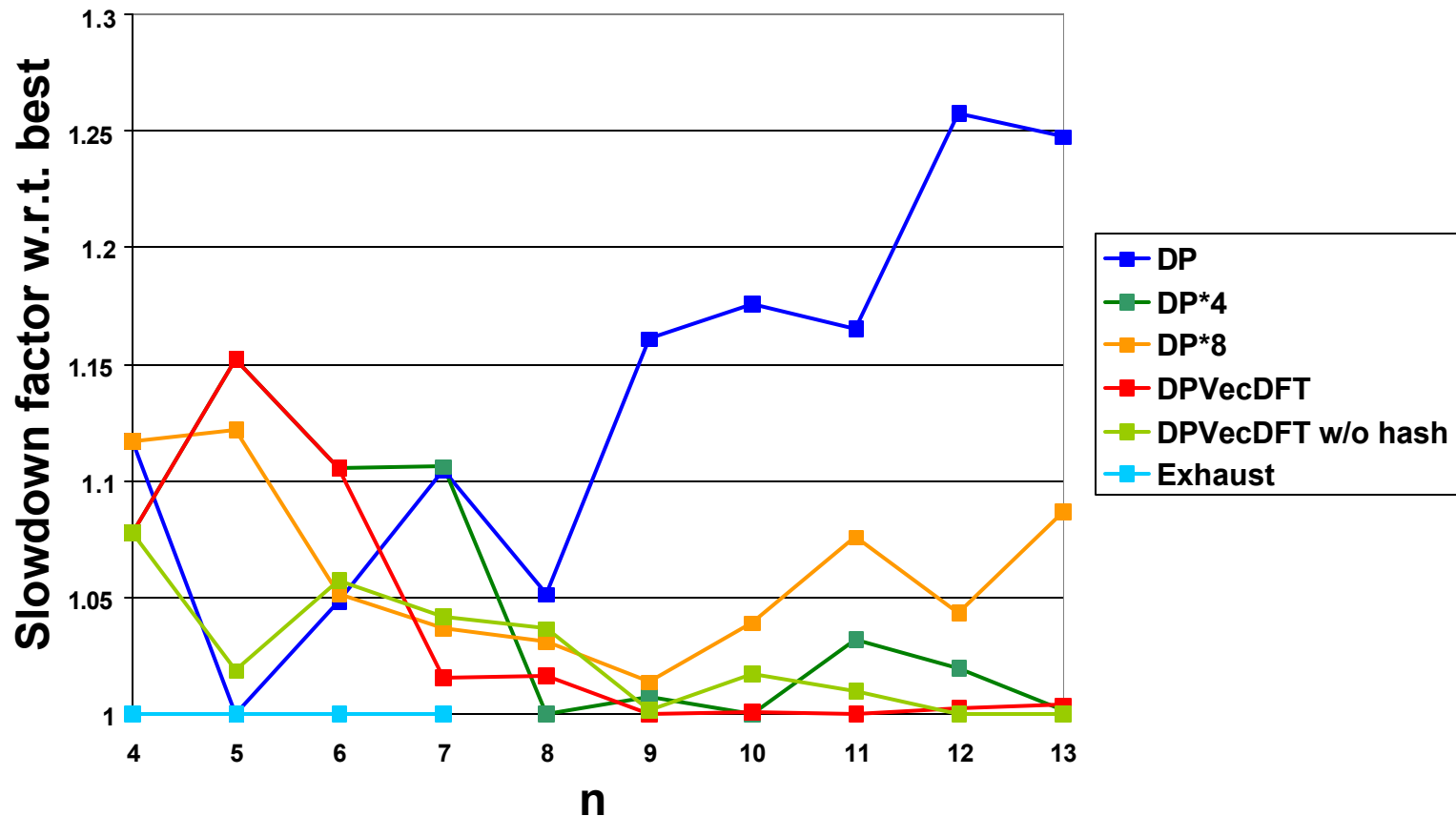


speedups (to C code) up to factor of 3



AURORA

Different search strategies



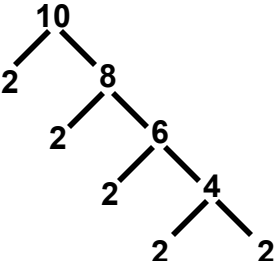
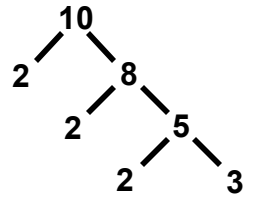
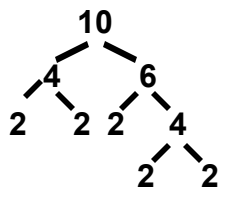
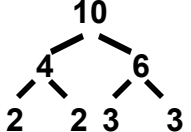
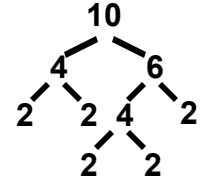
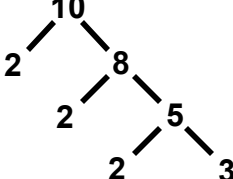
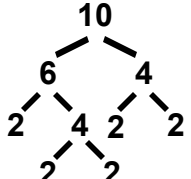
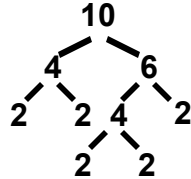
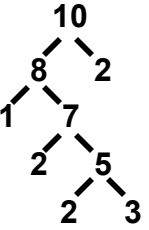
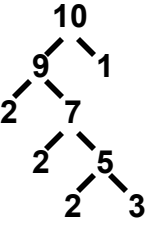
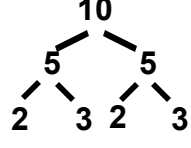
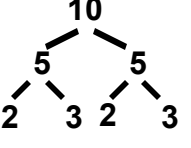
DFT 2^n single precision, Pentium 4, 2.53 GHz, using Intel C compiler 6.0



standard DP loses up to 25 % performance



Best DFT Trees, size $2^{10} = 1024$

	Pentium 4 float	Pentium 4 double	Pentium III float	AthlonXP float
scalar				
C vect				
SIMD				

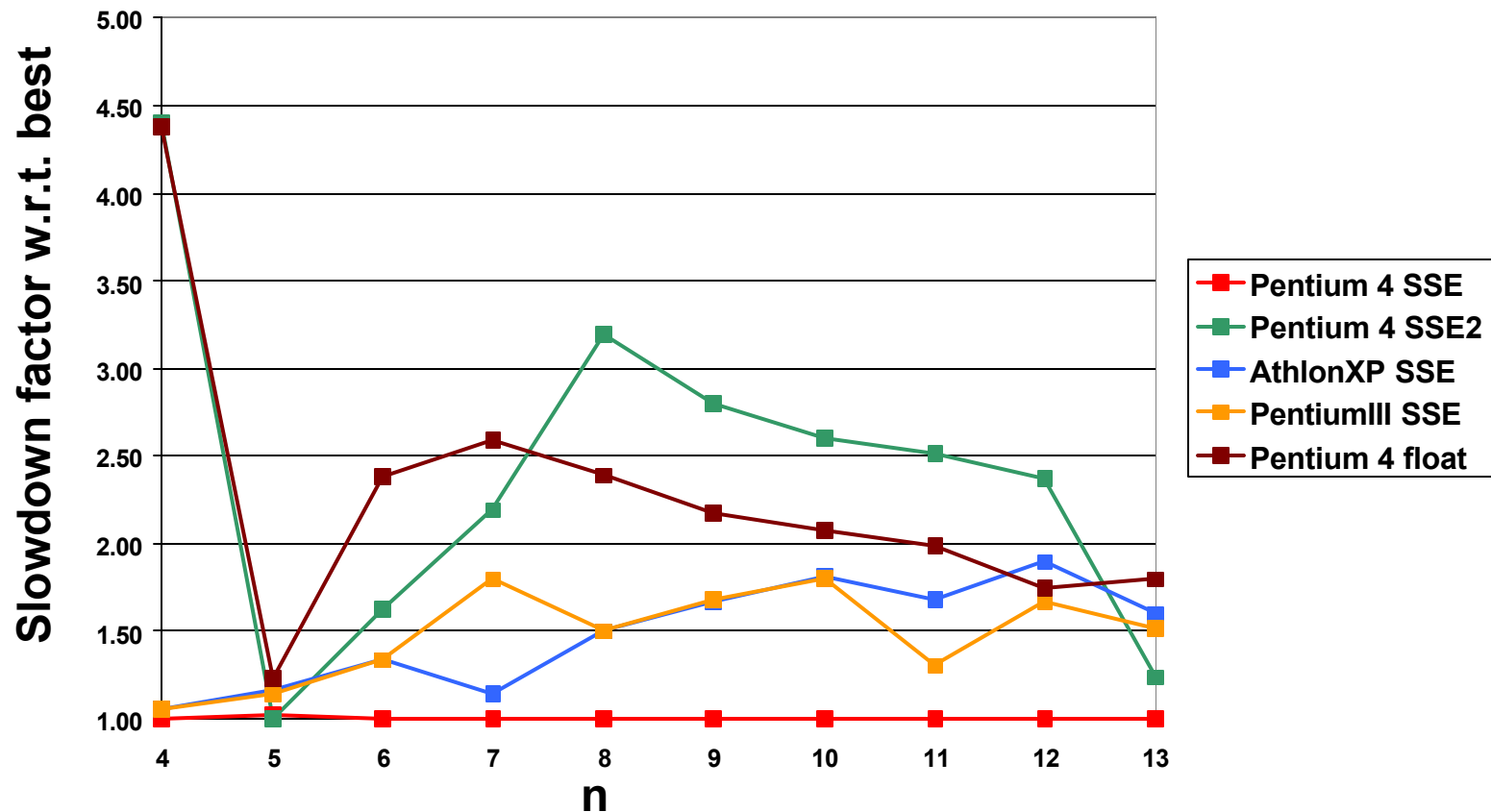


AURORA



trees platform/datatype dependent

Crosstiming of best trees on Pentium 4



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



software adaptation is necessary



Summary

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

Ongoing work:

- port to other SIMD architectures
- include filters and wavelets

<http://www.ece.cmu.edu/~spiral>

<http://www.math.tuwien.ac.at/~aurora>



AURORA