

Can We Teach Computers To Write Fast Libraries?

... and the Spiral team (only part shown)

Markus Püschel

Electrical and
Computer Engineering
Carnegie Mellon University

Joint work with
Franz Franchetti
Yevgen Voronenko

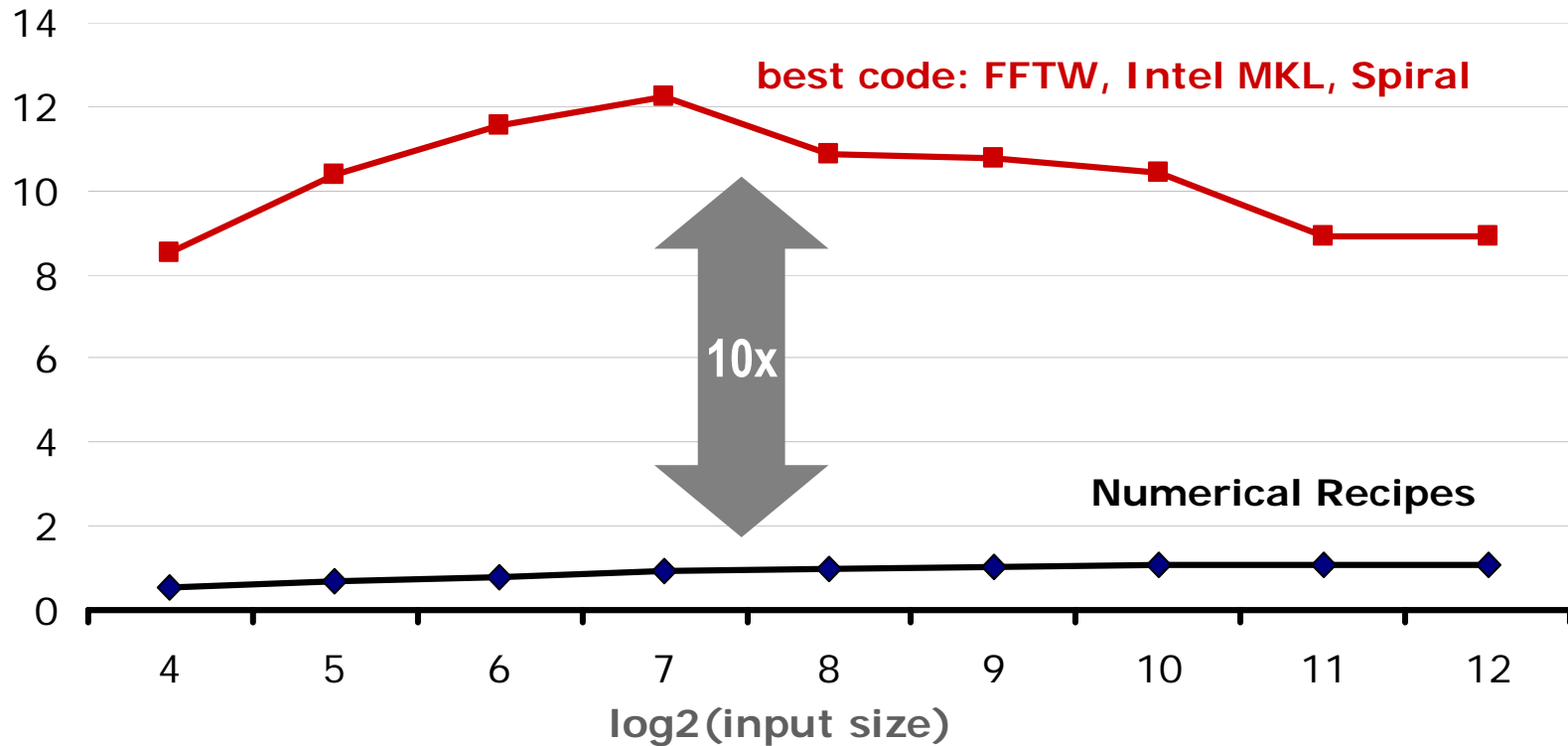


This work was supported by
DARPA DESA program, NSF-NGS/ITR, NSF-ACR, and Intel

The Problem

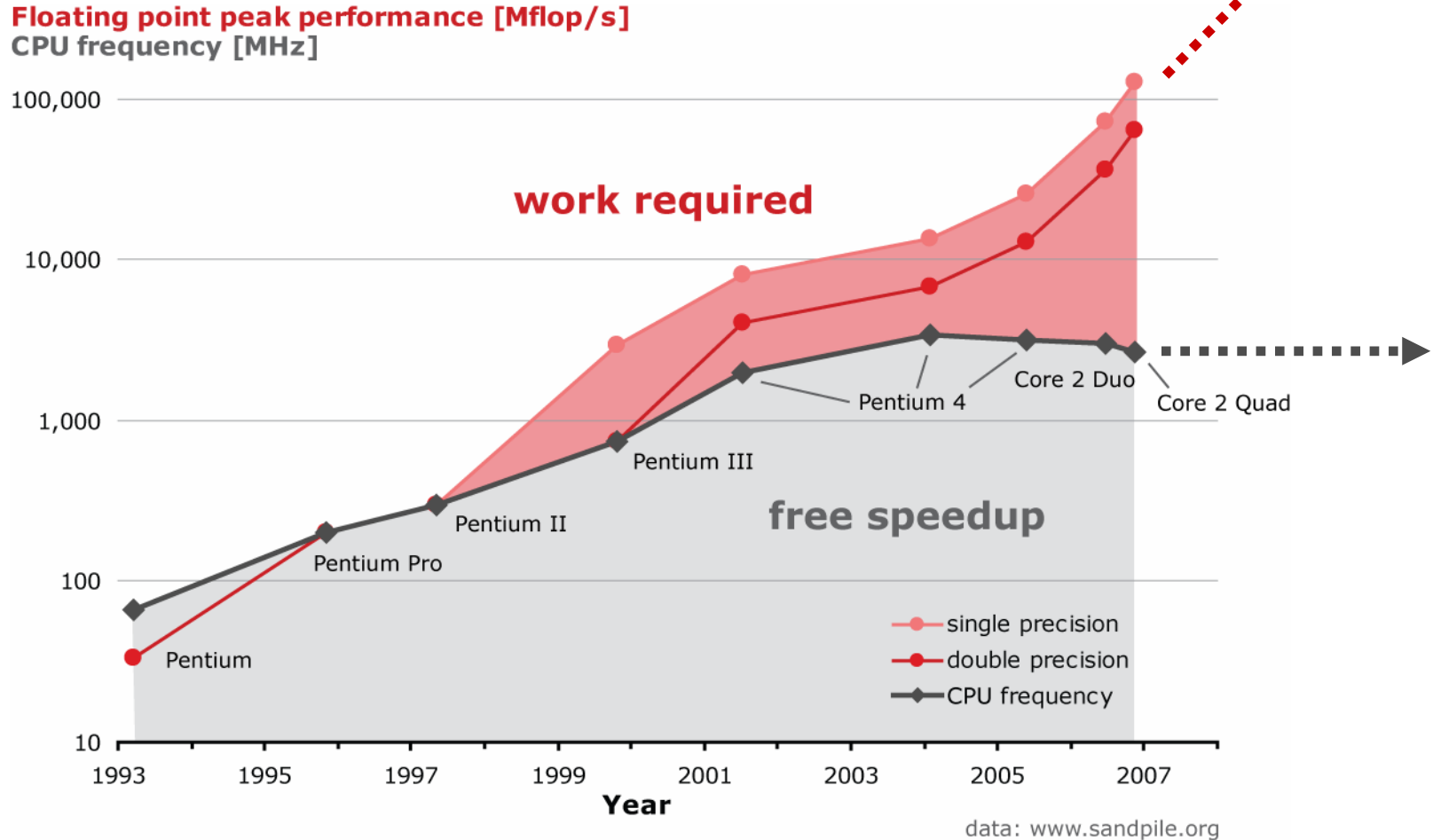
2.66 GHz Core 2 Duo
1 thread

Discrete Fourier transform (single precision)
performance [Gflop/s]



What's going on?

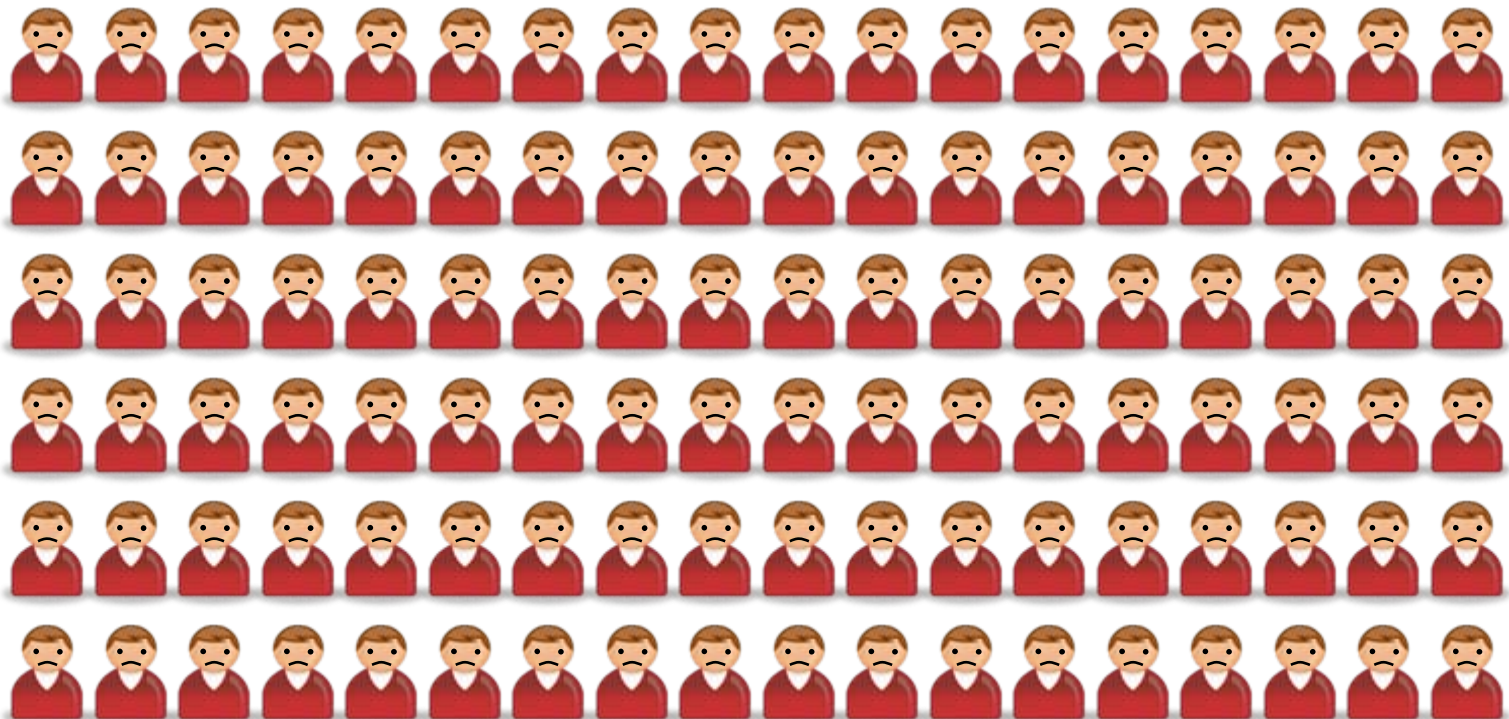
Evolution of Processors (Intel)



- High performance library development becomes increasingly difficult
- *What do we do?*

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

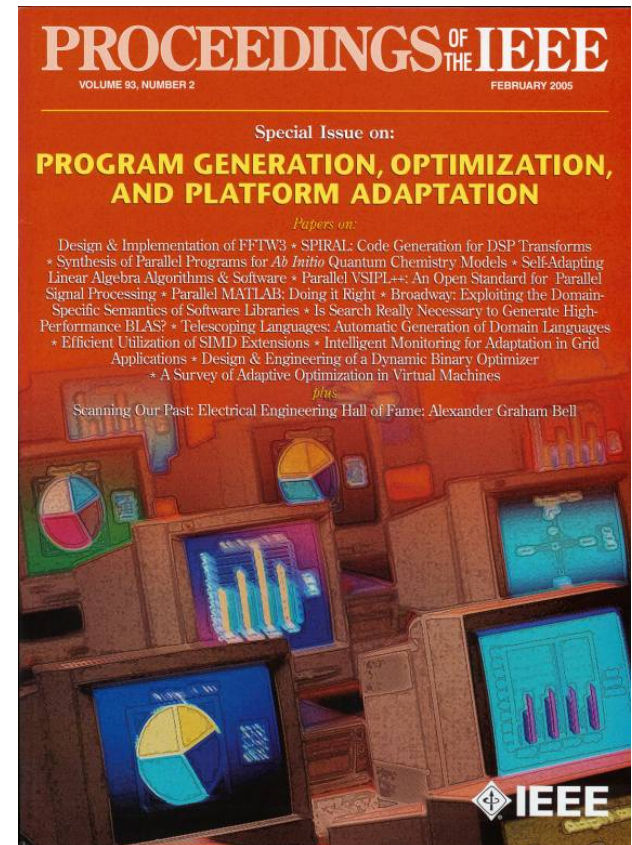
- Automate (parts of) the implementation or optimization



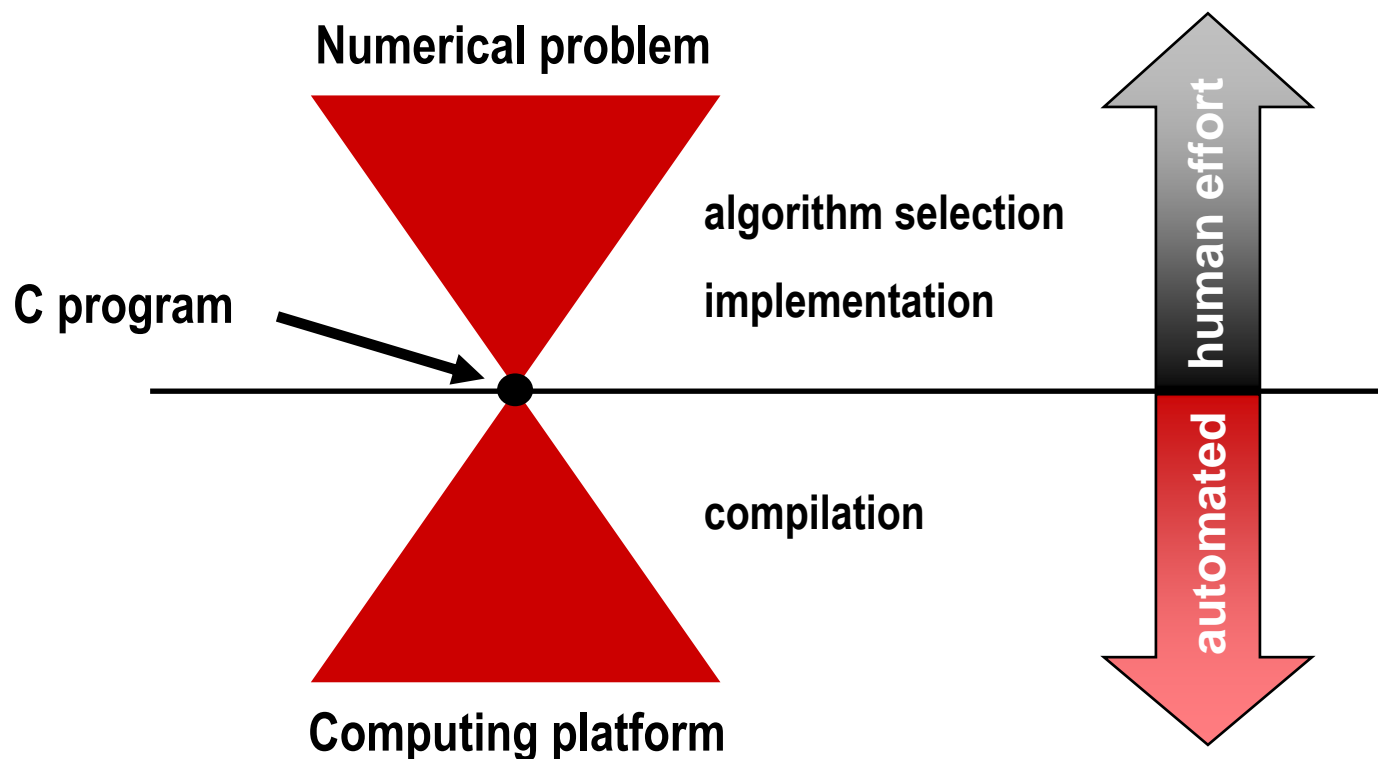
- **Research efforts**

- Linear algebra: Phipac/ATLAS, LAPACK, Sparsity/Bebop/OSKI, Flame
- Tensor computations
- PDE/finite elements: Fenics
- Adaptive sorting
- Fourier transform: FFTW
- Linear transforms: Spiral
- ...others
- New compiler techniques

**Promising new area but more work needed
In particular for parallelism ...**

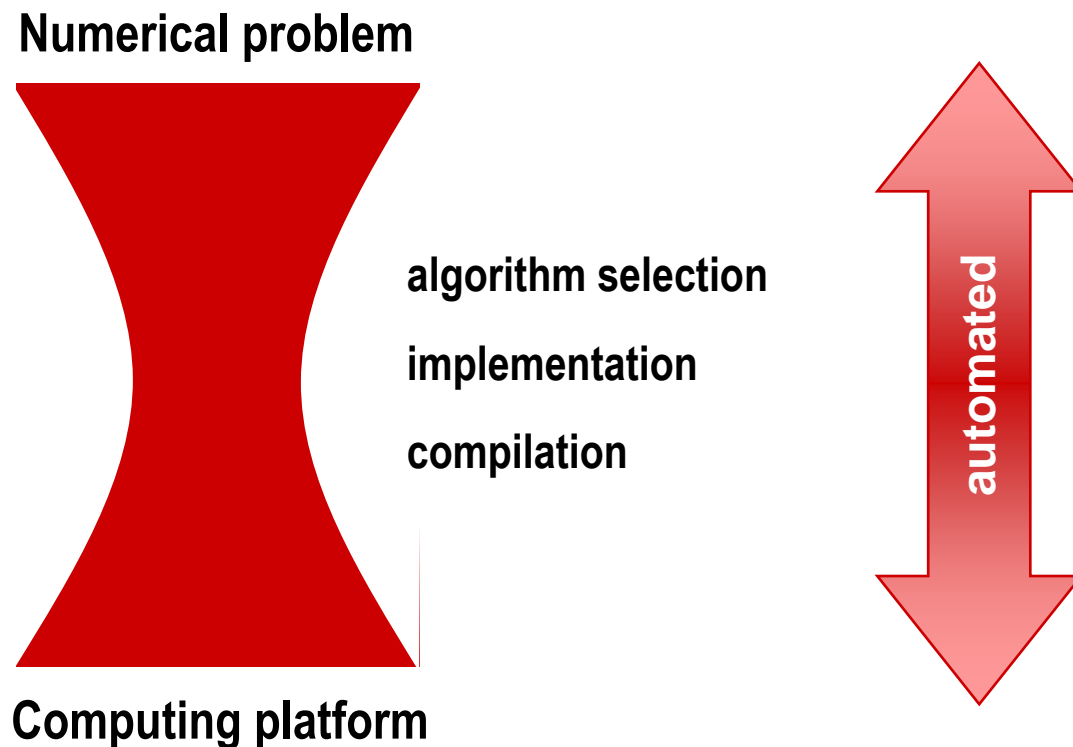


Current Approach



- Algorithm selection, implementation done by humans
- C code a “singularity:” compiler has no access to high level information

Future Approach?



- A new breed of program generation and optimization tools
- **Challenge:** Conquer the high abstraction level for automation

Organization

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

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 MKL library

How Spiral Works

Spiral:

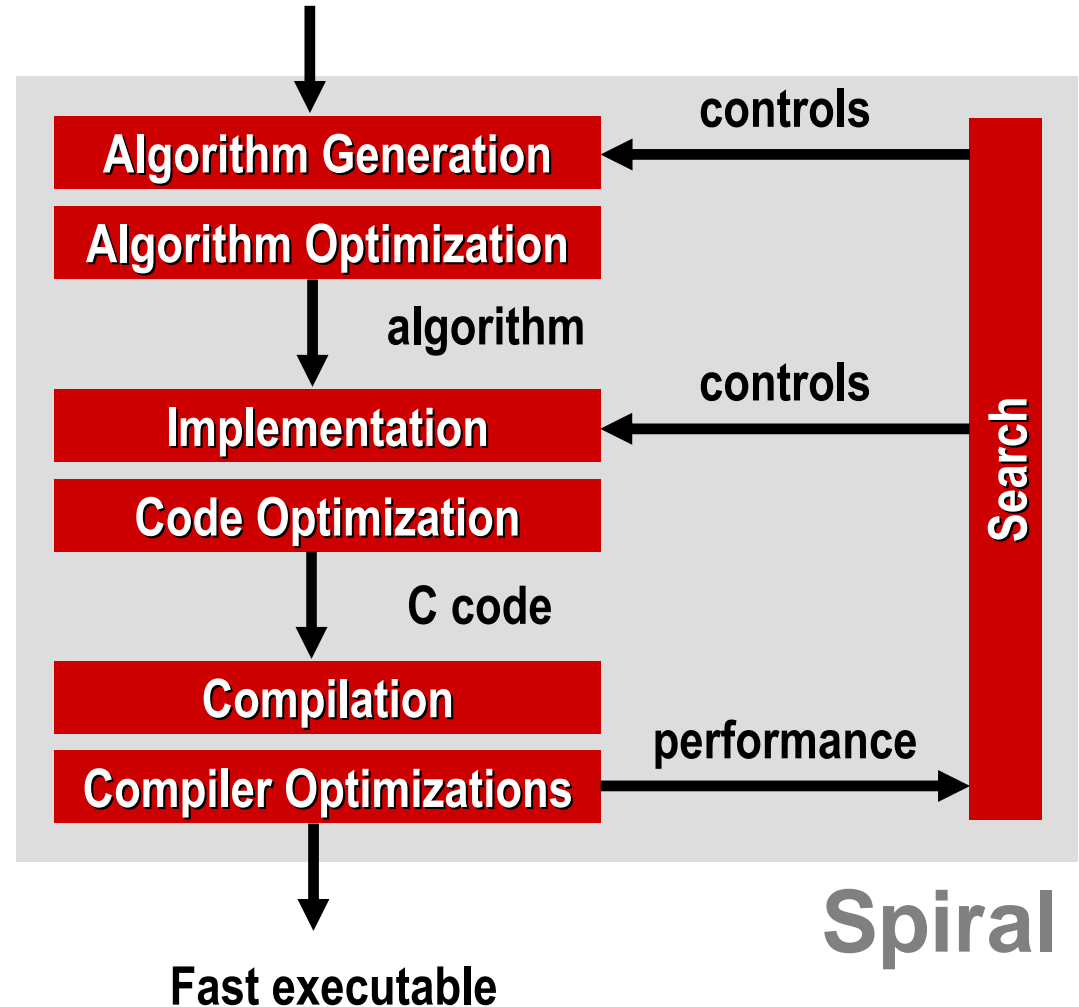
Complete automation of the implementation and optimization task

Basic ideas:

Declarative representation of algorithms

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

Problem specification (transform)



Web Interface (Beta Version)

Home
Code Generator
Publications
Software
Hardware
Grants
Team
Related Work
Internal

SPIRAL Program Generator for Transforms

What is it?

We provide a web interface to the SPIRAL program generator for linear transforms. You can select the platform, transform, transform size, and several other parameters, and SPIRAL will generate the C code for you, or retrieve it from the database if it was requested before. You can also browse the database below.

The interface is a beta version. This means minor bugs and reduced functionality. Soon, we will include the generation of vector code and parallel code (see the papers below).

You can download an earlier version of the program generator below.

We also provide interfaces to other generators (e.g., Verilog for transforms). See the list "Online Generators" on our [main page](#).

Known bugs: If you are using Internet Explorer: a) you will not be able to browse the archive; b) the generated C code will not have correct linebreaks. We are currently working to fix these bugs. Please, use Mozilla Firefox browser to avoid these problems.

Code Generator collapse

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

parameter	value	explanation
Transform	DCT2 (Discrete Cosine Transform, type 2)	The transform for which you want to request C code
Data type	double	The data type of input and output vector
Transform size	4	The size of the transform = the length of the input vector
Optimize for	runtime	What you want to optimize the code for
Search method	Dynamic Programming	The search method SPIRAL uses (Dynamic Programming is a good choice)
Compiler profile	gcc -O3	Compiler and compiler options used for compilation

Done

What is a (Linear) Transform?

- Mathematically: Matrix-vector multiplication

$$\begin{array}{c} \text{input vector (signal)} \quad x \mapsto y = T \cdot x \\ \text{output vector (signal)} \quad \uparrow \\ \text{transform = matrix} \end{array}$$

- Example: Discrete Fourier transform (DFT)

$$\text{DFT}_n = [e^{-2k\ell\pi i/n}]_{0 \leq k, \ell < n}$$

Transform Algorithms: Example 4-point FFT

Cooley/Tukey fast Fourier transform (FFT):

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix} = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & j \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

Fourier transform

Diagonal matrix (twiddles)

$$\text{DFT}_4 \rightarrow (\text{DFT}_2 \otimes \text{I}_2) \text{T}_2^4 (\text{I}_2 \otimes \text{DFT}_2) \text{L}_2^4$$

Kronecker product
Identity
Permutation

- 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)

$$\text{DCT-2}_n = \left[\cos(k(2l + 1)\pi/2n) \right]_{0 \leq k, l < n},$$

$$\text{DCT-3}_n = \text{DCT-2}_n^T \quad (\text{transpose}),$$

$$\text{DCT-4}_n = \left[\cos((2k + 1)(2l + 1)\pi/4n) \right]_{0 \leq k, l < n},$$

$$\text{IMDCT}_n = \left[\cos((2k + 1)(2l + 1 + n)\pi/4n) \right]_{0 \leq k < 2n, 0 \leq l < n},$$

$$\text{RDFT}_n = [r_{kl}]_{0 \leq k, l < n}, \quad r_{kl} = \begin{cases} \cos \frac{2\pi kl}{n}, & k \leq \lfloor \frac{n}{2} \rfloor \\ -\sin \frac{2\pi kl}{n}, & k > \lfloor \frac{n}{2} \rfloor \end{cases},$$

$$\text{WHT}_n = \begin{bmatrix} \text{WHT}_{n/2} & \text{WHT}_{n/2} \\ \text{WHT}_{n/2} & -\text{WHT}_{n/2} \end{bmatrix}, \quad \text{WHT}_2 = \text{DFT}_2,$$

$$\text{DHT} = \left[\cos(2kl\pi/n) + \sin(2kl\pi/n) \right]_{0 \leq k, l < n}.$$

Examples: Breakdown Rules (currently ≈220)

$$\text{DFT}_n \rightarrow (\text{DFT}_k \otimes \text{I}_m) \text{T}_m^n (\text{I}_k \otimes \text{DFT}_m) \text{L}_k^n, \quad n = km$$

$$\text{DFT}_n \rightarrow P_n (\text{DFT}_k \otimes \text{DFT}_m) Q_n, \quad n = km, \quad \text{gcd}(k, m) = 1$$

$$\text{DFT}_p \rightarrow R_p^T (\text{I}_1 \oplus \text{DFT}_{p-1}) D_p (\text{I}_1 \oplus \text{DFT}_{p-1}) R_p, \quad p \text{ prime}$$

$$\text{DCT-3}_n \rightarrow (\text{I}_m \oplus \text{J}_m) \text{L}_m^n (\text{DCT-3}_m(1/4) \oplus \text{DCT-3}_m(3/4))$$

$$\cdot (\text{F}_2 \otimes \text{I}_m) \begin{bmatrix} \text{I}_m & 0 \oplus -\text{J}_{m-1} \\ \frac{1}{\sqrt{2}}(\text{I}_1 \oplus 2\text{I}_m) \end{bmatrix}, \quad n = 2m$$

$$\text{DCT-4}_n \rightarrow S_n \text{DCT-2}_n \text{diag}_{0 \leq k < n} (1 / (2 \cos((2k + 1)\pi / 4n)))$$

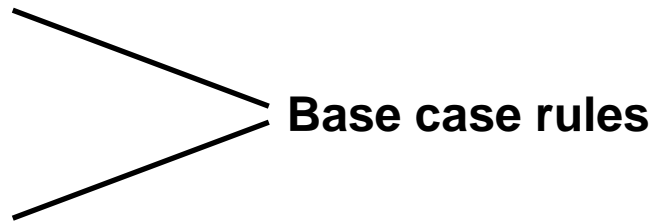
$$\text{IMDCT}_{2m} \rightarrow (\text{J}_m \oplus \text{I}_m \oplus \text{I}_m \oplus \text{J}_m) \left(\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \text{I}_m \right) \oplus \left(\begin{bmatrix} -1 \\ -1 \end{bmatrix} \otimes \text{I}_m \right) \right) \text{J}_{2m} \text{DCT-4}_{2m}$$

$$\text{WHT}_{2^k} \rightarrow \prod_{i=1}^t (\text{I}_{2^{k_1 + \dots + k_{i-1}}} \otimes \text{WHT}_{2^{k_i}} \otimes \text{I}_{2^{k_{i+1} + \dots + k_t}}), \quad k = k_1 + \dots + k_t$$

$$\text{DFT}_2 \rightarrow \text{F}_2$$

$$\text{DCT-2}_2 \rightarrow \text{diag}(1, 1/\sqrt{2}) \text{F}_2$$

$$\text{DCT-4}_2 \rightarrow \text{J}_2 \text{R}_{13\pi/8}$$



Combining these rules yields many algorithms for every given transform

Breakdown Rules (220)

BSkewDFT 4
 BSkewDFT_Base2
 BSkewDFT_Base4
 BSkewDFT_Fact
 BSkewDFT_Decom
 BSkewPRDFT 1
 BSkewPRDFT_Decom
 BSkewRDFT 1
 BSkewPRDFT_Decom
 Circulant 10
 RuleCirculant_Base
 RuleCirculant_toFilt
 RuleCirculant_Blocking
 RuleCirculant_BlockingDense
 RuleCirculant_DiagonalizeStep
 RuleCirculant_DFT
 RuleCirculant_RDFT
 RuleCirculant_PRDFT
 RuleCirculant_PRDFT1
 RuleCirculant_DHT
 CosDFT 2
 CosDFT_Base
 CosDFT_Trigonometric
 DCT1 3
 DCT1_Base2
 DCT1_Base4
 DCT1_DCT1and3
 DCT2 5
 DCT2_DCT2and4
 DCT2_toRDFT
 DCT2_toRDFT_odd
 DCT2_PrimePowerInduction
 DCT2_PrimeFactor
 DCT3 5
 DCT3_Base2
 DCT3_Base3
 DCT3_Base5
 DCT3_Orth9
 DCT3_toSkewDCT3
 DCT4 8
 DCT4_Base2
 DCT4_Base3
 DCT4_DCT2
 DCT4_DCT2t
 DCT4_DCT2andDST2
 DCT4_DST4andDST2
 DCT4_iterative
 DCT4_BSkew_Decom

DCT5 1
 DCT5_Rader
 DFT 22
 DFT_Base
 DFT_Canonicalize
 DFT_CT
 DFT_CT_MinCost
 DFT_CT_DDL
 DFT_CosSinSplit
 DFT_Rader
 DFT_Bluestein
 DFT_GoodThomas
 DFT_PFA_SUMS
 DFT_PFA_RaderComb
 DFT_SplitRadix
 DFT_DCT1andDST1
 DFT_DFT1and3

Cooley-Tukey FFT

DRDFT 1
 DRDFT_tSPL_Pease
 DSCirculant 2
 RuleDSCirculant_Base
 DST1_Base2
 DST1_DST3and1
 DST2 3
 DST2_Base2
 DST2_toDCT2
 DST2_DST2and4
 DST4 2
 DST4_Base
 DST4_toDCT4
 DTT 15

Filt 10
 RuleFilt_Base
 RuleFilt_Nest
 RuleFilt_OverlapSave
 RuleFilt_OverlapSaveFreq
 RuleFilt_OverlapAdd
 RuleFilt_OverlapAdd2
 RuleFilt_KaratSubaSimple
 RuleFilt_Circulant
 RuleFilt_Blocking
 RuleFilt_KaratSubaFast
 DFT 5
 IPRDFT1_Base1
 IPRDFT1_Base2
 IPRDFT1_CT
 IPRDFT1_Complex
 IPRDFT1_CT_Radix2
 IPRDFT2 4
 IPRDFT2_Base1
 IPRDFT2_Base2
 IPRDFT2_CT
 IPRDFT2_CT_Radix2
 IPRDFT3 3

PDHT2 2
 PDHT2_Base2
 PDHT2_CT
 PDHT3 4
 PDHT3_Base2
 PDHT3_CT
 PDHT3_Trig
 PDHT3_CT_Radix2
 PDHT4 3
 PDHT4_Base2
 PDHT4_CT
 PDHT4_Trig
 PDST4 2
 PDST4_Base2
 PDST4_CT
 PRDFT 10
 PRDFT1_Base1
 PRDFT1_Base2
 PRDFT1_CT
 PRDFT1_Complex
 PRDFT1_Complex_T
 PRDFT1_Trig
 PRDET1_PE

PolyDFT 4
 PolyDFT_ToNormal
 PolyDFT_Base2
 PolyDFT_SkewBase2
 PolyDFT_SkewDST3_CT
 RDFT 4
 RDFT_Base
 RDFT_Trigonometric
 RDFT_CT_Radix2
 RDFT_toDCT2
 RHT 2
 RHT_Base
 RHT_CooleyTukey
 SinDFT 2
 SinDFT_Base
 SinDFT_Trigonometric
 SkewDFT 2
 SkewDFT_Base2
 SkewDFT_Fact
 SkewDFT 3
 SkewDFT_Base2
 SkewDCT3_VarSteidl
 SkewDCT3_VarSteidlIterative
 Toeplitz 6
 RuleToeplitz_Base
 RuleToeplitz_SmartBase
 RuleToeplitz_PreciseBase
 RuleToeplitz_Blocking
 RuleToeplitz_BlockingDense

RuleToeplitz_ExpandedConvolution
 UDFT_Base4
 UDFT_SR_Reg
 URDFT 4
 URDFT1_Base1
 URDFT1_Base2
 URDFT1_Base4
 URDFT1_CT
 WHT 8
 WHT_Base
 WHT_GeneralSplit
 WHT_BinSplit
 WHT_DDL
 WHT_Dirsum
 WHT_tSPL_BinSplit
 WHT_tSPL_STerm
 WHT_tSPL_Pease

**Breakdown rules in Spiral =
 Knowledge about existing transform algorithms
 (~50 journal papers)**

DFTBR 1
 DFTBR_HW
 DHT 3
 DHT_Base
 DHT_DCT1andDST1
 DHT_Radix2

DWTper 4
 DWTper_Mallat_2
 DWTper_Mallat
 DWTper_Polyphase
 DWTper_Lifting

PDCT4_CT
 PDHT 3
 PDHT1_Base2
 PDHT1_CT
 PDHT1_CT_Radix2

PolyBDFT 5
 PolyBDFT_Base2
 PolyBDFT_Base4
 PolyBDFT_Fact
 PolyBDFT_Trig
 PolyBDFT_Decom

A Word About These Rules

■ Algebraic Theory of Transform Algorithms

- Use (abstract) algebra to derive, explain, and understand transform algorithms
- Reduces > 100 papers to a small number of principles
- Enables the derivation of new algorithms
- More info: www.ece.cmu.edu/~smart

■ In some cases, automatic algorithm discovery is possible (AREP, Egner/Pueschel 97/98)

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \cos(\frac{1}{16}\pi) & \cos(\frac{3}{16}\pi) & \cos(\frac{5}{16}\pi) & \cos(\frac{7}{16}\pi) & \cos(\frac{9}{16}\pi) & \cos(\frac{11}{16}\pi) & \cos(\frac{13}{16}\pi) & \cos(\frac{15}{16}\pi) \\ \cos(\frac{1}{8}\pi) & \cos(\frac{3}{8}\pi) & \cos(\frac{5}{8}\pi) & \cos(\frac{7}{8}\pi) & \cos(\frac{9}{8}\pi) & \cos(\frac{11}{8}\pi) & \cos(\frac{13}{8}\pi) & \cos(\frac{15}{8}\pi) \\ \cos(\frac{3}{16}\pi) & \cos(\frac{9}{16}\pi) & \cos(\frac{15}{16}\pi) & \cos(\frac{11}{16}\pi) & \cos(\frac{5}{16}\pi) & \cos(\frac{1}{16}\pi) & \cos(\frac{7}{16}\pi) & \cos(\frac{13}{16}\pi) \\ \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} \\ \cos(\frac{5}{16}\pi) & \cos(\frac{15}{16}\pi) & \cos(\frac{7}{16}\pi) & \cos(\frac{3}{16}\pi) & \cos(\frac{13}{16}\pi) & \cos(\frac{9}{16}\pi) & \cos(\frac{1}{16}\pi) & \cos(\frac{11}{16}\pi) \\ \cos(\frac{3}{8}\pi) & \cos(\frac{1}{8}\pi) & \cos(\frac{5}{8}\pi) & \cos(\frac{7}{8}\pi) & \cos(\frac{9}{8}\pi) & \cos(\frac{11}{8}\pi) & \cos(\frac{13}{8}\pi) & \cos(\frac{15}{8}\pi) \\ \cos(\frac{7}{16}\pi) & \cos(\frac{11}{16}\pi) & \cos(\frac{3}{16}\pi) & \cos(\frac{15}{16}\pi) & \cos(\frac{1}{16}\pi) & \cos(\frac{13}{16}\pi) & \cos(\frac{5}{16}\pi) & \cos(\frac{17}{16}\pi) \end{bmatrix}$$



$$\begin{aligned} & [(2, 5)(4, 7)(6, 8), 8] \cdot \\ & (\mathbf{1}_2 \oplus R_{\frac{3}{8}\pi} \oplus R_{\frac{15}{16}\pi} \oplus R_{\frac{21}{16}\pi}) \cdot \\ & [(2, 4, 7, 3, 8), (1, 1, 1, \sqrt{\frac{1}{2}}, 1, 1, 1, 1)] \cdot \\ & ((\text{DFT}_2 \otimes \mathbf{1}_3) \oplus \mathbf{1}_2) \cdot \\ & [(5, 6), 8] \cdot \\ & (\mathbf{1}_4 \oplus \sqrt{\frac{1}{2}} \cdot \text{DFT}_2 \oplus \mathbf{1}_2) \cdot \\ & [(2, 3, 4, 5, 8, 6, 7), 8] \cdot \\ & (\mathbf{1}_2 \otimes ((\text{DFT}_2 \oplus \mathbf{1}_2) \cdot [(2, 3), 4] \cdot (\mathbf{1}_2 \otimes \text{DFT}_2))) \cdot \\ & [(1, 8, 6, 2)(3, 4, 5, 7), 8] \end{aligned}$$

SPL to Sequential Code

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

Example: tensor product

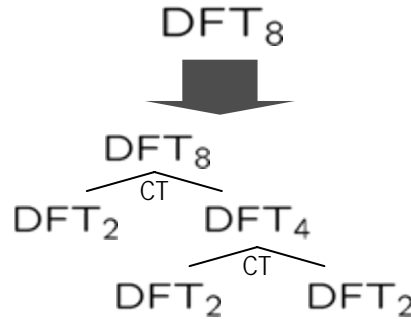
$$I_m \otimes A_n = \begin{bmatrix} A_n & & \\ & \dots & \\ & & A_n \end{bmatrix}$$

Correct code: easy fast code: very difficult

Program Generation (Simplified)

Transform:

Ruletree:



$$DFT_8 \rightarrow (DFT_2 \otimes I_4) \cdot D \cdot (I_2 \otimes DFT_4) \cdot P$$

$$DFT_4 \rightarrow (DFT_2 \otimes I_2) \cdot D \cdot (I_2 \otimes DFT_2) \cdot P$$

SPL Formula:

$$(DFT_2 \otimes I_4) T_4^8 \left(I_2 \otimes \left((DFT_2 \otimes I_2) T_2^4 (I_2 \otimes DFT_2) L_2^4 \right) \right) L_2^8$$

$$\sum_{j=0}^3 (S_j DFT_2 G_j) \sum_{k=0}^1 \left(\sum_{l=0}^1 (S_{k,l} \text{diag}(t_{k,l}) DFT_2 G_l) \sum_{m=0}^1 (S_m \text{diag}(t_m) DFT_2 G_{k,m}) \right)$$

C Code:

```
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;
}
```

**Parallelization at the
high abstraction level**

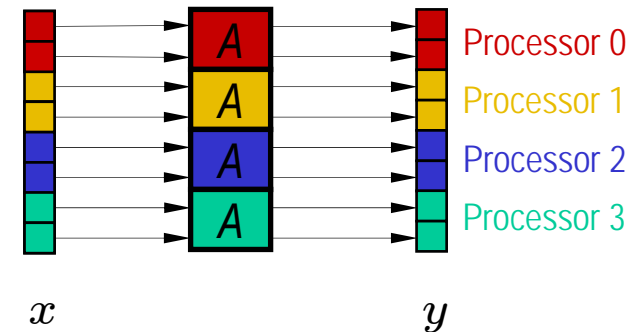
Organization

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

SPL to Shared Memory Code: Basic Idea

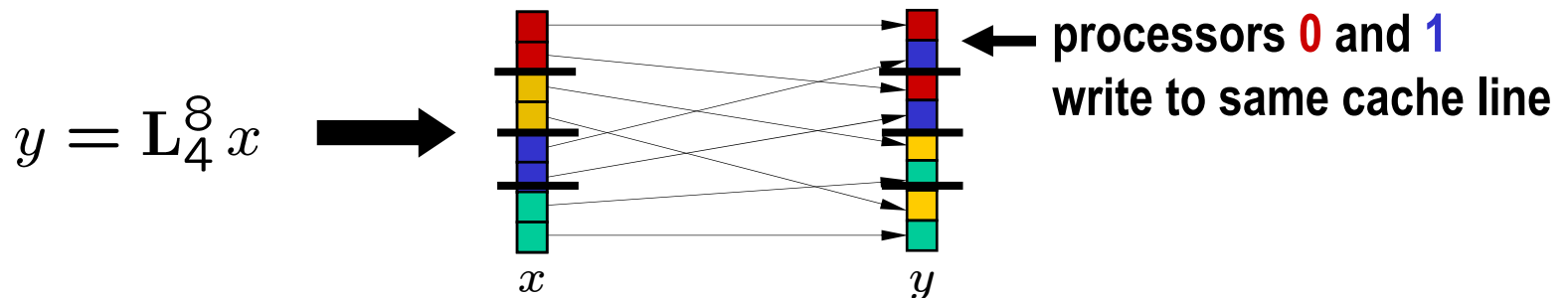
- Good construct: tensor product

$$y = \left(I_p \otimes A \right) x$$



p-way embarrassingly parallel, load-balanced

- Problematic construct: permutations produce false sharing



Task: Rewrite formulas to extract tensor product + make cache lines atomic

Step 1: Shared Memory Tags

- Identify crucial hardware parameters = coarse hardware abstraction

- Number of processors: p
- Cache line size: μ

- Introduce them as tags in SPL:

$$\overset{A}{\text{smp}(p, \mu)}$$

This means: Formula A is to be optimized for p processors and cache line size μ

Step 2: Identify “Good” Formulas

- Load balanced, avoiding false sharing

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

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

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

- **Definition:** A formula is **fully optimized** for p processors and cacheline size μ 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 based on tags
- There may be choices

$$\begin{aligned}
 \underbrace{AB}_{\text{smp}(p,\mu)} &\rightarrow \underbrace{A}_{\text{smp}(p,\mu)} \underbrace{B}_{\text{smp}(p,\mu)} \\
 \underbrace{A_m \otimes I_n}_{\text{smp}(p,\mu)} &\rightarrow \underbrace{\left(L_m^{mp} \otimes I_{n/p} \right) \left(I_p \otimes (A_m \otimes I_{n/p}) \right) \left(L_p^{mp} \otimes I_{n/p} \right)}_{\text{smp}(p,\mu)} \\
 \underbrace{L_m^{mn}}_{\text{smp}(p,\mu)} &\rightarrow \begin{cases} \underbrace{\left(I_p \otimes L_{m/p}^{mn/p} \right)}_{\text{smp}(p,\mu)} \underbrace{\left(L_p^{pn} \otimes I_{m/p} \right)}_{\text{smp}(p,\mu)} \\ \underbrace{\left(L_m^{pm} \otimes I_{n/p} \right)}_{\text{smp}(p,\mu)} \underbrace{\left(I_p \otimes L_m^{mn/p} \right)}_{\text{smp}(p,\mu)} \end{cases} \\
 \underbrace{I_m \otimes A_n}_{\text{smp}(p,\mu)} &\rightarrow I_p \otimes_{\parallel} \left(I_{m/p} \otimes A_n \right) \\
 \underbrace{(P \otimes I_n)}_{\text{smp}(p,\mu)} &\rightarrow \left(P \otimes I_{n/\mu} \right) \bar{\otimes} I_\mu
 \end{aligned}$$

Simple Rewriting Example

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p, \mu)}$$



Loop tiling and scheduling
hardware-conscious (knows p and μ)

$$\left(L_m^{mp} \otimes I_{n/p} \right) \left(I_p \otimes_{||} (A_m \otimes I_{n/p}) \right) \left(L_p^{mp} \otimes I_{n/p} \right)$$

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]);
```

Example: DFT

Hardware parameters: p processors, cache line length μ

$$\underbrace{\text{DFT}_{mn}}_{\text{smp}(p,\mu)} \rightarrow \underbrace{\left((\text{DFT}_m \otimes \text{I}_n) \text{T}_n^{mn} (\text{I}_m \otimes \text{DFT}_n) \text{L}_m^{mn} \right)}_{\text{smp}(p,\mu)}$$

...

$$\rightarrow \underbrace{(\text{DFT}_m \otimes \text{I}_n)}_{\text{smp}(p,\mu)} \underbrace{\text{T}_n^{mn}}_{\text{smp}(p,\mu)} \underbrace{(\text{I}_m \otimes \text{DFT}_n)}_{\text{smp}(p,\mu)} \underbrace{\text{L}_m^{nm}}_{\text{smp}(p,\mu)}$$

...

$$\rightarrow \underbrace{\left((\text{L}_m^{mp} \otimes \text{I}_{n/p\mu}) \bar{\otimes} \text{I}_\mu \right)}_{\text{red}} \underbrace{\left(\text{I}_p \otimes_{\parallel} (\text{DFT}_m \otimes \text{I}_{n/p}) \right)}_{\text{blue}} \underbrace{\left((\text{L}_p^{mp} \otimes \text{I}_{n/p\mu}) \bar{\otimes} \text{I}_\mu \right)}_{\text{red}}$$

$$\underbrace{\left(\bigoplus_{i=0}^{p-1} \text{T}_n^{mn,i} \right)}_{\text{blue}} \underbrace{\left(\text{I}_p \otimes_{\parallel} (\text{I}_{m/p} \otimes \text{DFT}_n) \right)}_{\text{blue}} \underbrace{\left(\text{I}_p \otimes_{\parallel} \text{L}_{m/p}^{mn/p} \right)}_{\text{blue}} \underbrace{\left((\text{L}_p^{pn} \otimes \text{I}_{m/p\mu}) \bar{\otimes} \text{I}_\mu \right)}_{\text{red}}$$

Fully optimized (**load-balanced**, **no false sharing**)
in the sense of our definition

Same Approach for Other Parallel Paradigms

Message Passing: [ISPA 06]

$$\begin{aligned}
 \underbrace{\text{DFT}_{mn}}_{\text{par}(p)} &\rightarrow \underbrace{(\text{DFT}_m \otimes \text{I}_n)}_{\text{par}(p \leftarrow q)} \underbrace{\text{T}_n^{mn}}_{\text{par}(q)} \underbrace{(\text{I}_m \otimes \text{DFT}_n)}_{\text{par}(q)} \underbrace{\text{L}_m^{mn}}_{\text{par}(q \leftarrow p)} \\
 &\dots \\
 &\dots \\
 &\rightarrow \underbrace{(\text{I}_p \otimes \text{L}_{m/p}^{mn/p})}_{\text{par}(p)} \underbrace{(\text{L}_p^{p^2} \otimes \text{I}_{mn/p^2})}_{\text{par}(p \leftarrow q)} \underbrace{(\text{I}_q \otimes (\text{I}_{p/q} \otimes \text{L}_p^n \otimes \text{I}_{m/p}))}_{\text{par}(q)} \underbrace{(\text{I}_q \otimes (\text{I}_{n/q} \otimes \text{DFT}_m))}_{\text{par}(q)} \\
 &\dots \\
 &\rightarrow \underbrace{(\text{I}_q \otimes \text{L}_{m/q}^{mn/q})}_{\text{par}(q)} \underbrace{(\text{L}_q^{q^2} \otimes \text{I}_{mn/q^2})}_{\text{par}(q)} \underbrace{(\text{I}_q \otimes (\text{L}_q^n \otimes \text{I}_{m/q}))}_{\text{par}(q)} \underbrace{\text{T}_n^{mn}}_{\text{par}(q)} \underbrace{(\text{I}_q \otimes (\text{I}_{m/q} \otimes \text{DFT}_n))}_{\text{par}(q)} \\
 &\dots \\
 &\rightarrow \underbrace{(\text{I}_q \otimes (\text{I}_{p/q} \otimes \text{L}_{m/p}^{mn/p}))}_{\text{par}(q)} \underbrace{(\text{L}_p^{p^2} \otimes \text{I}_{mn/p^2})}_{\text{par}(q \leftarrow p)} \underbrace{(\text{I}_p \otimes (\text{L}_p^n \otimes \text{I}_{m/p}))}_{\text{par}(p)}
 \end{aligned}$$

With Bonelli, Lorenz, Ueberhuber, TU Vienna

Vectorization: [IPDPS 02, Vecpar 06]

$$\begin{aligned}
 \underbrace{(\text{DFT}_{mn})}_{\text{vec}(\nu)} &\rightarrow \underbrace{(\text{DFT}_m \otimes \text{I}_n) \text{T}_n^{mn} (\text{I}_m \otimes \text{DFT}_n) \text{L}_m^{mn}}_{\text{vec}(\nu)} \\
 &\dots \\
 &\rightarrow \underbrace{(\text{DFT}_m \otimes \text{I}_n)}_{\text{vec}(\nu)}^\nu \underbrace{(\text{T}_n^{mn})}_{\text{vec}(\nu)}^\nu \underbrace{(\text{I}_m \otimes \text{DFT}_n) \text{L}_m^{mn}}_{\text{vec}(\nu)}^\nu \\
 &\dots \\
 &\rightarrow (\text{I}_{mn/\nu} \otimes \underbrace{\text{L}_\nu^{2\nu}}_{\text{sse}}) (\text{DFT}_m \otimes \text{I}_{n/\nu} \vec{\text{I}}_\nu) (\underbrace{\text{T}_n^{mn}}_{\text{sse}})^\nu \\
 &\quad (\text{I}_{m/\nu} \otimes (\underbrace{\text{L}_\nu^n \vec{\otimes} \text{I}}_\nu) (\text{I}_{n/\nu} \otimes (\underbrace{\text{L}_\nu^{2\nu} \vec{\otimes} \text{I}}_\nu) (\text{I}_2 \otimes \underbrace{\text{L}_\nu^{2\nu}}_{\text{sse}}) (\text{L}_2^{2\nu} \vec{\otimes} \text{I}_\nu))) (\text{DFT}_n \vec{\otimes} \text{I}_\nu) \\
 &\quad (\underbrace{(\text{L}_m^{mn} \otimes \text{I}_2) \vec{\otimes} \text{I}}_\nu) (\text{I}_{mn/\nu} \otimes \underbrace{\text{L}_2^{2\nu}}_{\text{sse}})
 \end{aligned}$$

Cg/OpenGL for GPUs:

$$\begin{aligned}
 \underbrace{(\text{DFT}_{r^k})}_{\text{gpu}(t,c)} &\rightarrow \underbrace{\left(\prod_{i=0}^{k-1} \text{L}_r^{r^k} (\text{I}_{r^{k-1}} \otimes \text{DFT}_r) (\text{L}_{r^{k-i-1}} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}) \text{L}_{r^{i+1}}^{r^k}) \right)}_{\text{gpu}(t,c)} \text{R}_r^{r^k} \\
 &\dots \\
 &\rightarrow \left(\prod_{i=0}^{k-1} (\text{L}_r^{r^n/2} \vec{\otimes} \text{I}_2) (\text{I}_{r^{n-1}/2} \otimes \times \underbrace{(\text{DFT}_r \vec{\otimes} \text{I}_2) \text{L}_r^{2r}}_{\text{shd}(t,c)}) \text{T}_i \right) \\
 &\quad (\text{L}_r^{r^n/2} \vec{\otimes} \text{I}_2) (\text{I}_{r^{n-1}/2} \otimes \times \underbrace{\text{L}_r^{2r}}_{\text{shd}(t,c)}) (\text{R}_r^{r^{n-1}} \vec{\otimes} \text{I}_r)
 \end{aligned}$$

With Shen, TU Denmark

Verilog for FPGAs: [DAC 05]

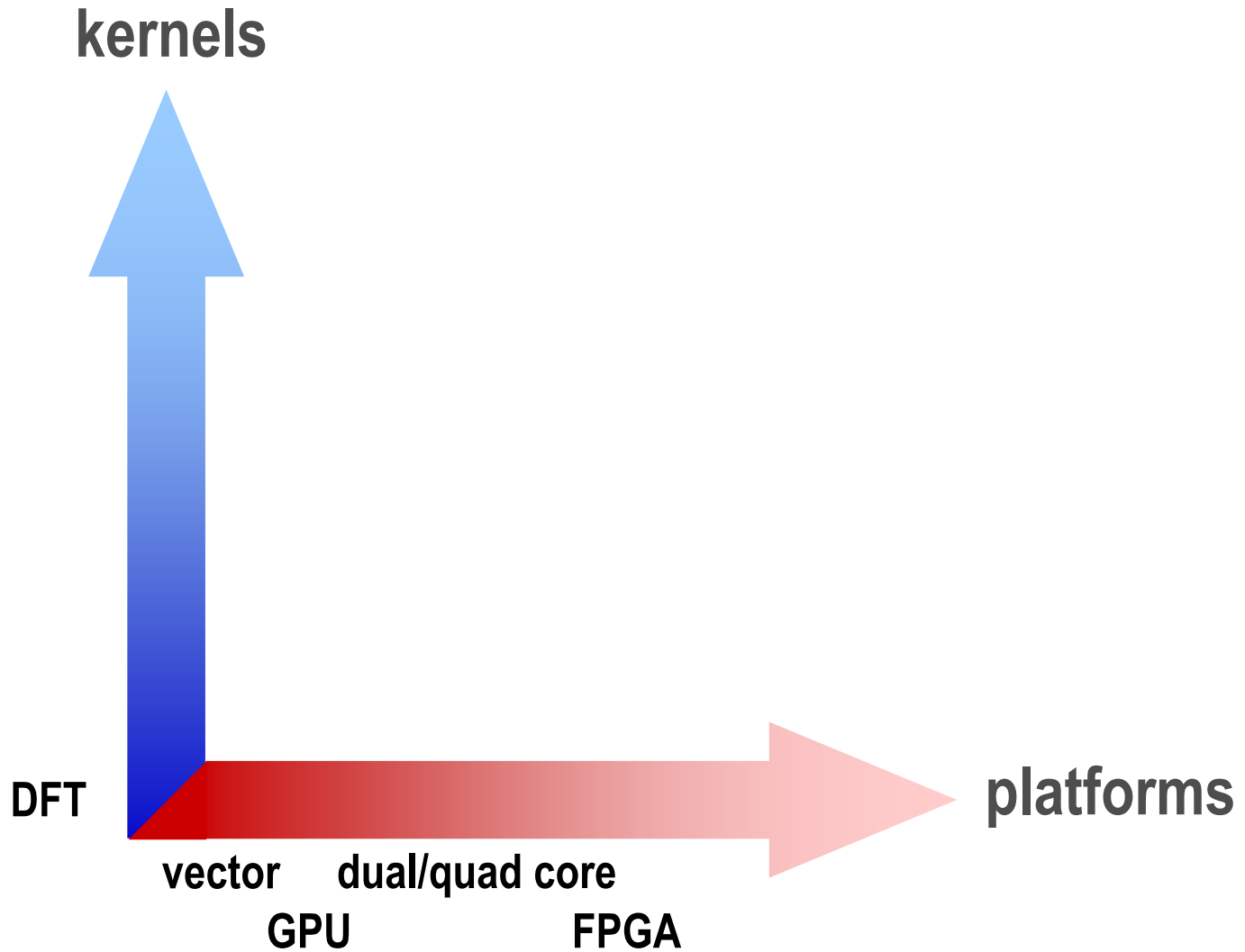
$$\begin{aligned}
 \underbrace{(\text{DFT}_{r^k})}_{\text{stream}(r^s)} &\rightarrow \underbrace{\left[\prod_{i=0}^{k-1} \text{L}_r^{r^k} (\text{I}_{r^{k-1}} \otimes \text{DFT}_r) (\text{L}_{r^{k-i-1}} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}) \text{L}_{r^{i+1}}^{r^k}) \right]}_{\text{stream}(r^s)} \text{R}_r^{r^k} \\
 &\dots \\
 &\rightarrow \left[\prod_{i=0}^{k-1} \underbrace{\text{L}_r^{r^k}}_{\text{stream}(r^s)} \underbrace{(\text{I}_{r^{k-1}} \otimes \text{DFT}_r)}_{\text{stream}(r^s)} \underbrace{(\text{L}_{r^{k-i-1}} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}) \text{L}_{r^{i+1}}^{r^k})}_{\text{stream}(r^s)} \right]_{\text{stream}(r^s)} \text{R}_r^{r^k} \\
 &\dots \\
 &\rightarrow \left[\prod_{i=0}^{k-1} \underbrace{\text{L}_r^{r^k}}_{\text{stream}(r^s)} (\text{I}_{r^{k-s-1}} \otimes_s (\text{I}_{r^{s-1}} \otimes \text{DFT}_r)) \underbrace{\text{T}_i'}_{\text{stream}(r^s)} \right]_{\text{stream}(r^s)} \text{R}_r^{r^k}
 \end{aligned}$$

With Milder, Hoe, CMU

Organization

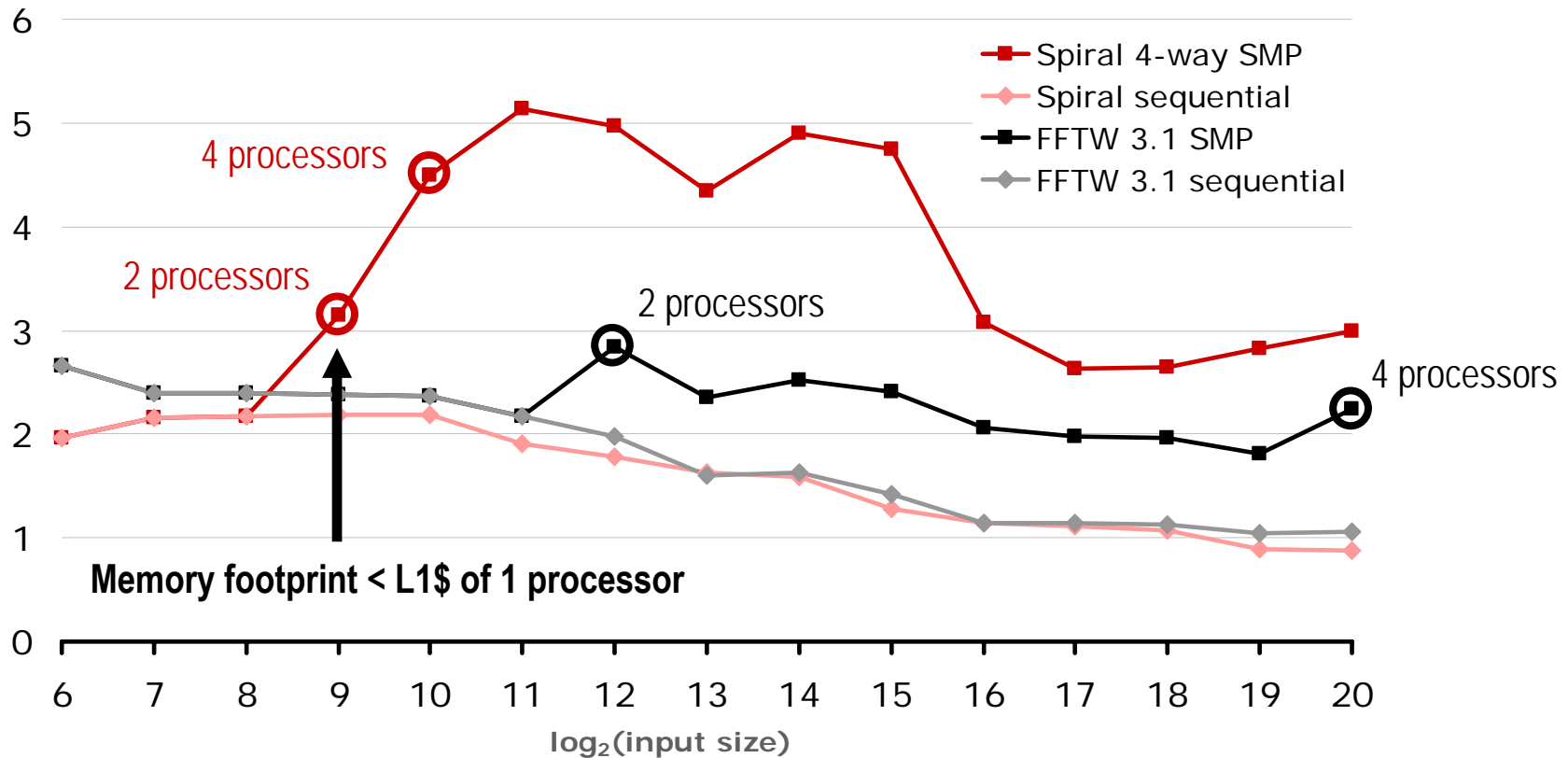
- Spiral: Brief overview
- Parallelization in Spiral
- **Results**
- Conclusion

Benchmarks



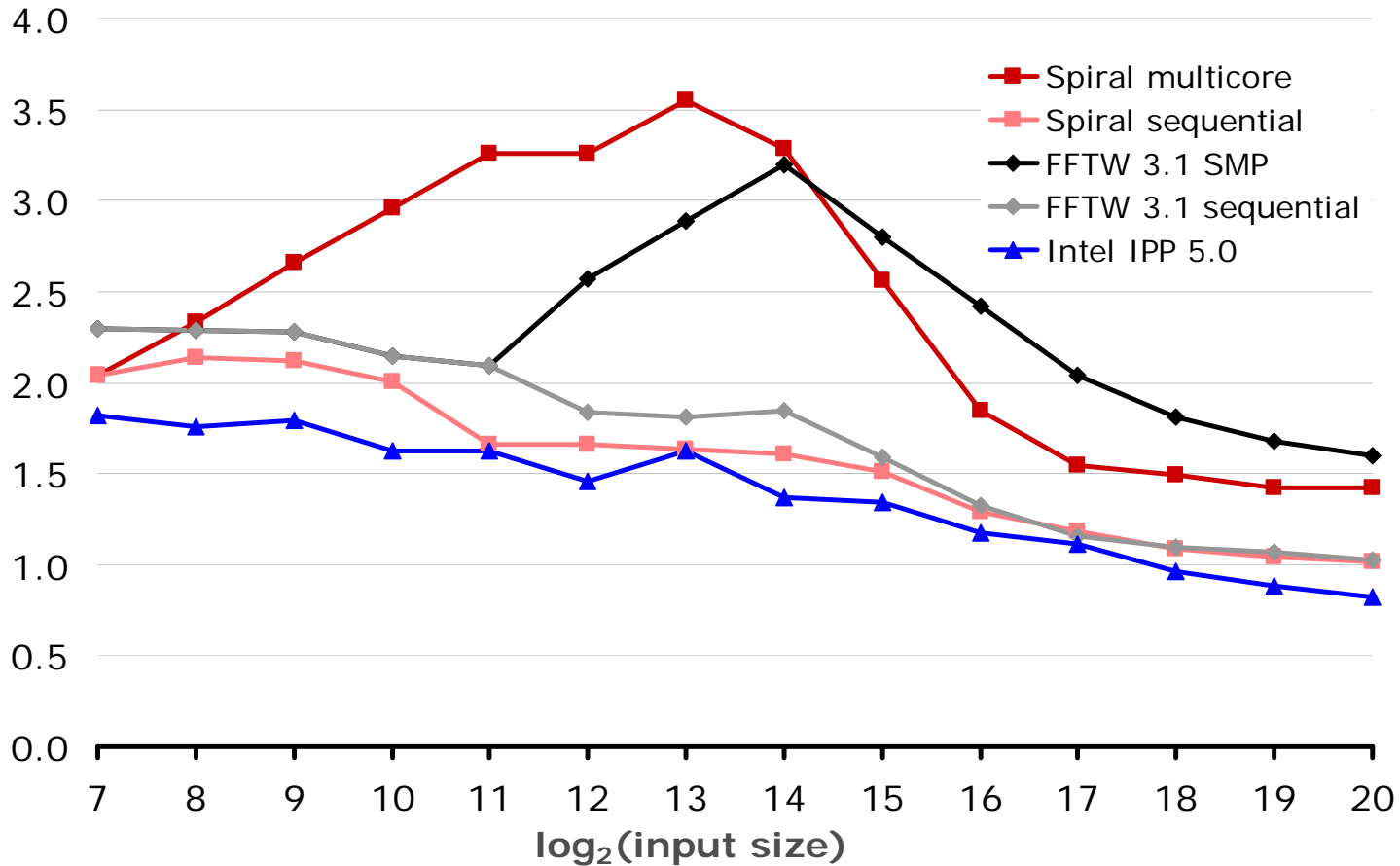
Benchmark: Vector And SMP

DFT (double precision) on 2.2 GHz Dual Opteron Dualcore (4 processors)
 performance [Gflop/s]



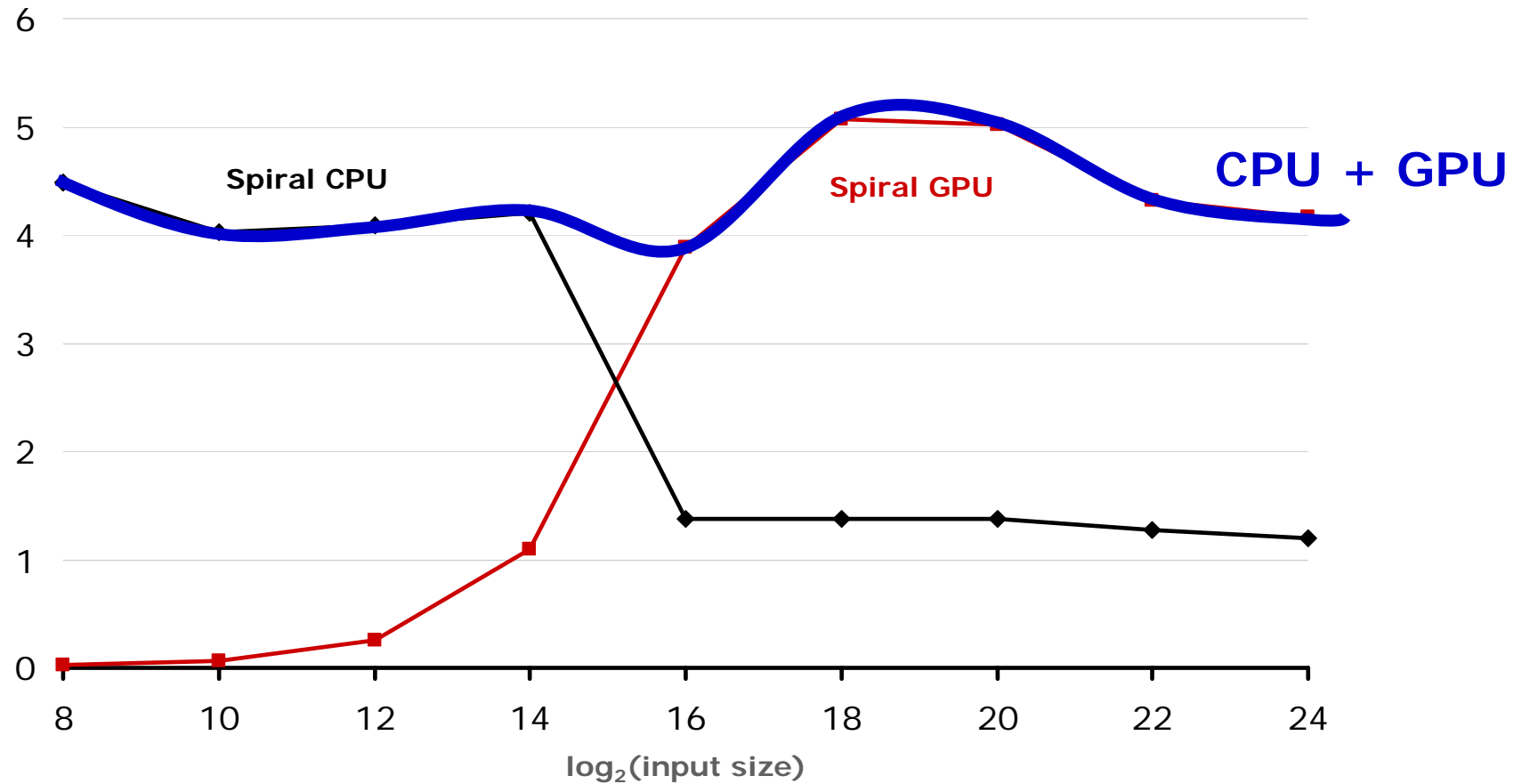
Benchmark: Vector and SMP

DFT (double precision) on 2.0 GHz Core Duo (2 Processors)
 performance [Gflop/s]



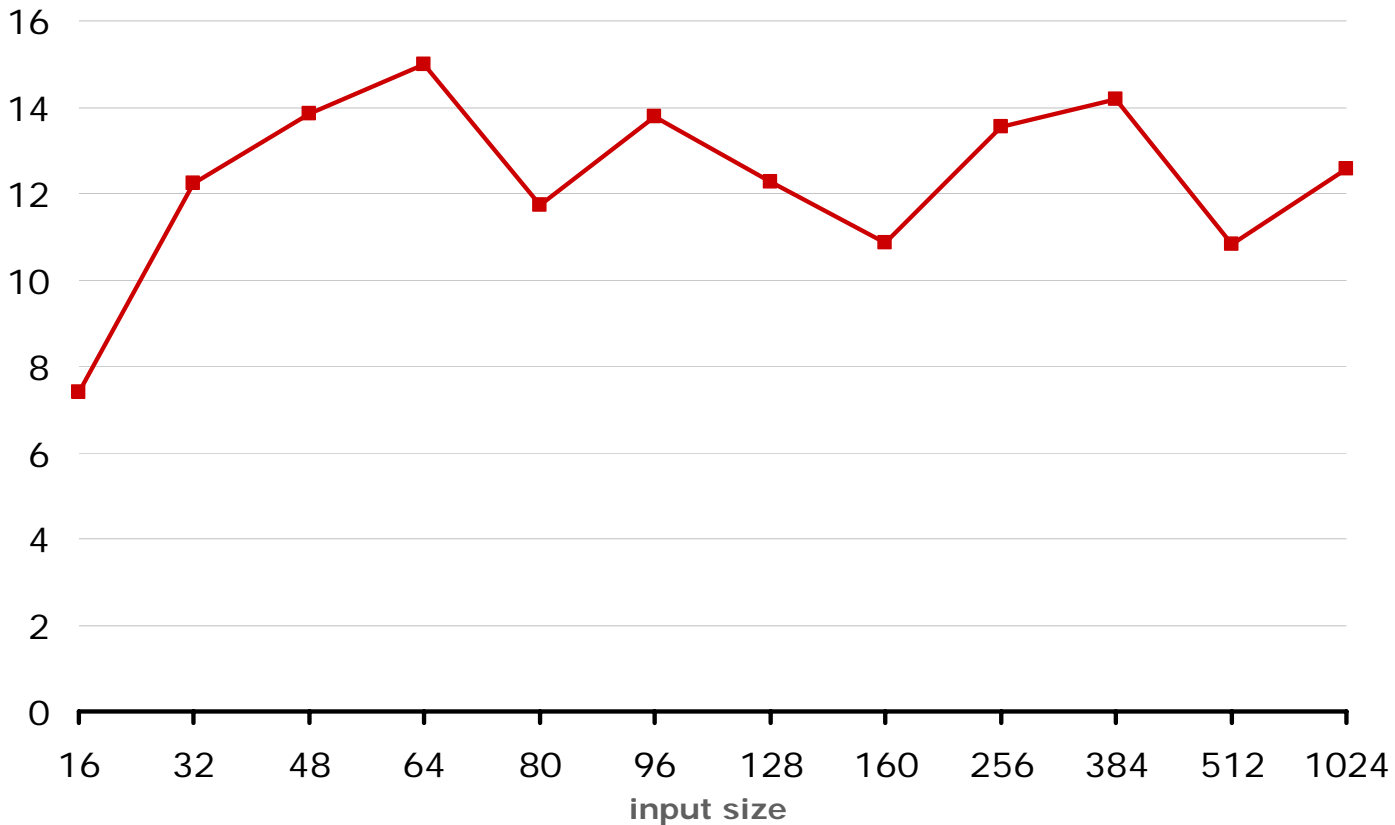
Benchmark: GPU

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



Benchmark: Cell (1 processor = SPE)

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

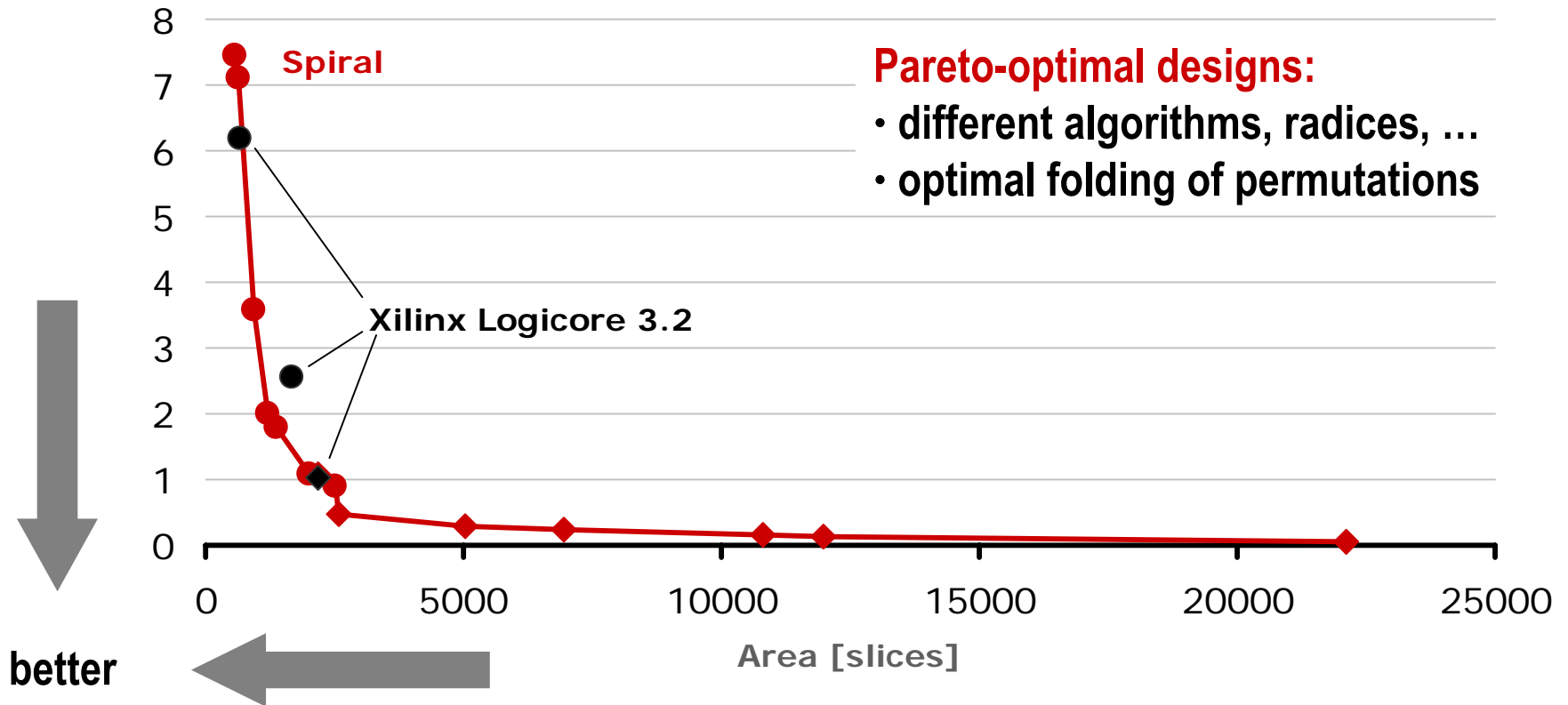


Generated using the simulator; run at Mercury (thanks to Robert Cooper)

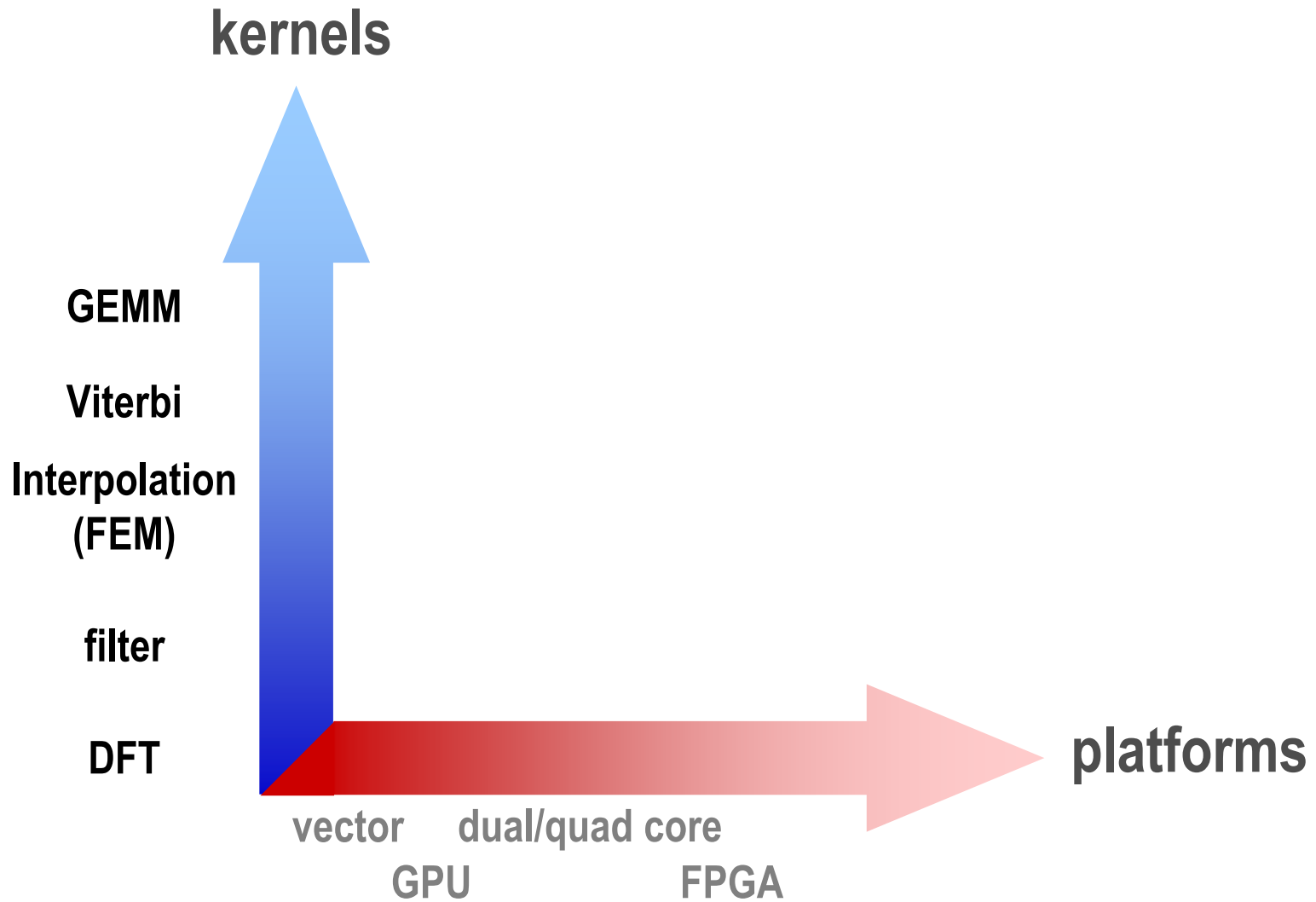
Joint work with Th. Peter (ETH Zurich), M. Telgarsky, J. Moura (CMU)

Benchmark: FPGA

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

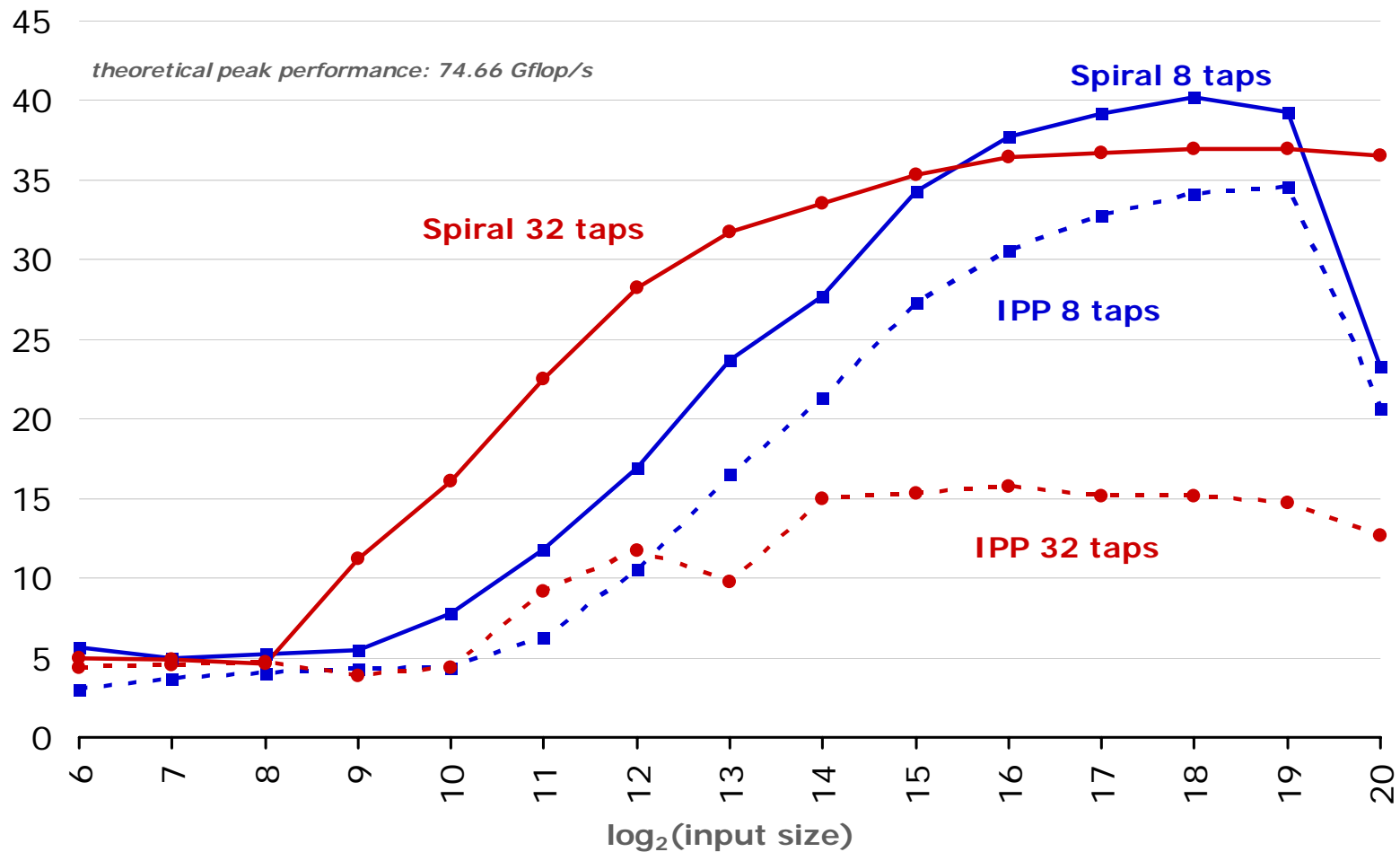


Benchmarks



Benchmark: Finite Impulse Response Filter

FIR filter (double precision) on 2.33 GHz 2x Core 2 Quad (8 threads)
 performance [Gflop/s]



Interpolation in Finite Element Methods

Original matrix:

```
[[0.22360679775,0.22360679775,0.22360679775,-0.67082039325],
[0.0,0.0,0.0,0.0],
[-0.22360679775,0.67082039325,-0.22360679775,-0.22360679775],
[0.0,0.0,0.0,0.0],
[1.1708203933,0.27639320225,0.27639320225,0.27639320225],
[0.0,-0.894427191,0.0,0.894427191],
[-1.1708203933,-0.27639320225,-0.27639320225,-0.27639320225],
[-0.27639320225,-0.27639320225,-1.1708203933,-0.27639320225],
[0.0,0.0,0.0,0.0],
[0.27639320225,0.27639320225,1.1708203933,0.27639320225]];
```



real DFT further expanded by Spiral

$$\begin{aligned}
 & [(2, 8, 6, 4, 9, 10, 7, 5, 3), 10] \cdot \mathbf{1}_{(10,7)} \cdot \\
 & [(1, 2)(3, 4, 7)(5, 6), (1, 1, 1, 1, -1, -1, 1)] \cdot \\
 & (F_2 \oplus \text{RDFT}_4 \oplus \mathbf{1}_1) \cdot [(1, 4, 6)(5, 7), 7] \cdot \mathbf{1}_{(7,5)} \cdot \\
 & (0.44721359552500006 \cdot \mathbf{1}_1 \oplus \\
 & \quad \left[\begin{array}{cc} -0.22360679775 & 0.22360679774999997 \\ 0.72360679777500003 & 0.27639320225000003 \end{array} \right] \oplus \\
 & \quad \left[\begin{array}{c} -0.44721359550000001 \\ 0.89442719100000001 \end{array} \right]) \cdot \\
 & [(1, 2)(3, 4), 4] \cdot (F_2 \oplus F_2) \cdot [(1, 3, 4, 2), (1, -1, 1, 1)]
 \end{aligned}$$

double-loop: **30a + 40m, 237 ns**
 unrolled + CSE (Spiral): **13a + 14m, 113 ns**

try this by hand

```
void sub(double *Y, double *X) {
    double t3032, t3033, t3034, t3035, t3036, t3037, t3038, t3039, t3040;
    t3032 = (X[2] + X[0]);
    t3033 = (0.44721359552500006*(X[2] - X[0]));
    t3034 = (X[3] - X[1]);
    t3035 = (0.44721359550000001*t3034);
    t3036 = (X[3] + X[1]);
    t3037 = ((0.22360679775*t3036) - (0.22360679775*t3032));
    t3038 = ((0.72360679777500003*t3032) + (0.27639320225000003*t3036));
    t3039 = (t3038 - t3033);
    t3040 = (t3038 + t3033);
    Y[0] = -((t3035 + t3037));
    Y[1] = 0;
    Y[2] = (t3037 - t3035);
    Y[3] = 0;
    Y[4] = t3039;
    Y[5] = (0.89442719100000001*t3034);
    Y[6] = -(t3039);
    Y[7] = -(t3040);
    Y[8] = 0;
    Y[9] = t3040;
}
```



10a + 7m, 67 ns

Going Beyond Transforms

- Transform =
 - linear** operator with **one** vector input and **one** vector output
- Key idea:
 - 1) Generalize to (**possibly nonlinear**) operators with **several** inputs and **several** outputs
 - 2) Generalize SPL (including tensor product) to OL (operator language)

Cooley-Tukey FFT in OL: $DFT \rightarrow (DFT \otimes I) \circ D \circ (I \otimes DFT) \circ L.$

Viterbi in OL: $Vit \rightarrow \pi \circ \left(\prod (I \otimes V) \circ (L \times I) \right) \circ (C \times C \times I)$

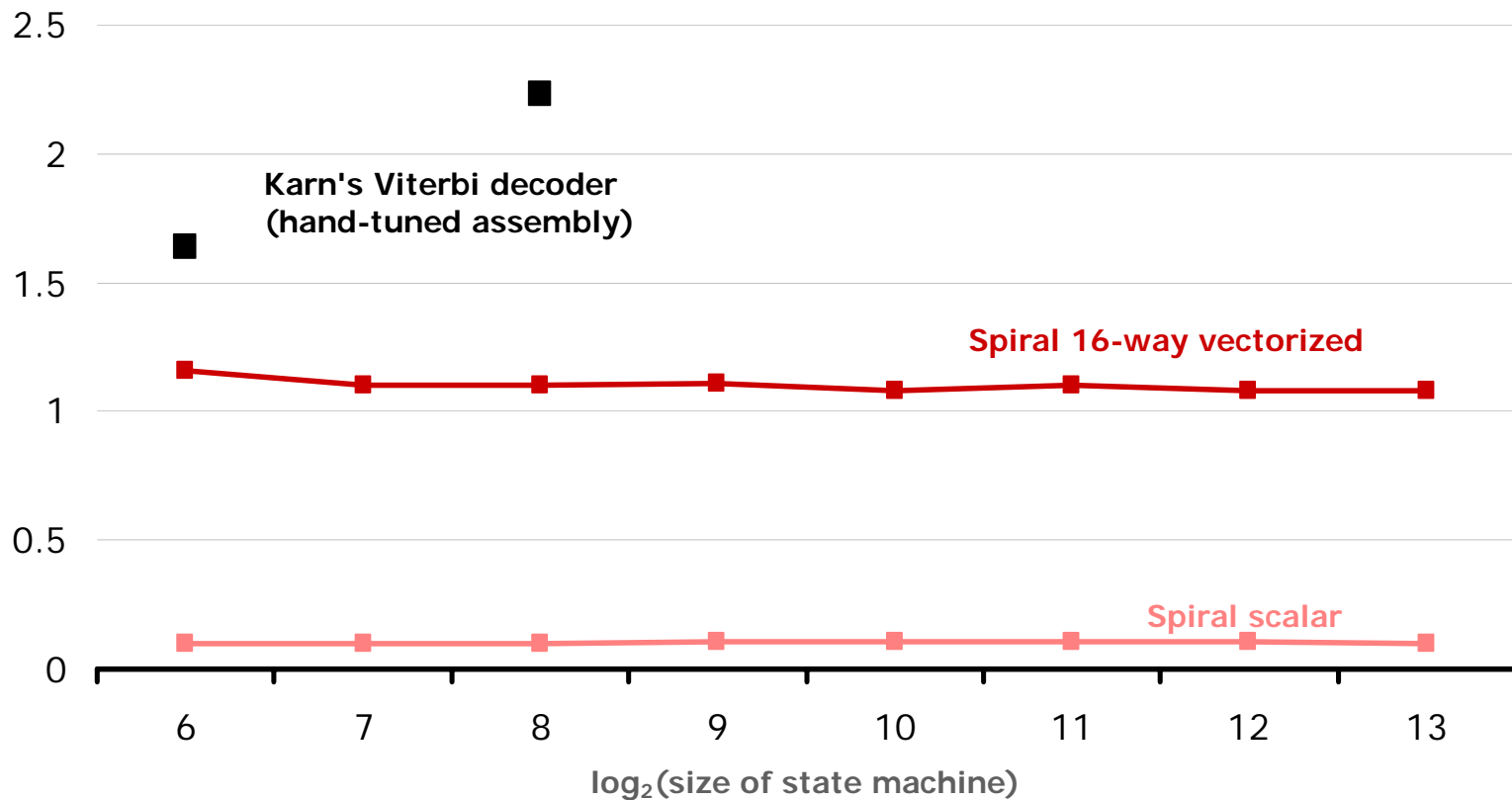
Mat-Mat-Mult: $MMM \rightarrow I \otimes MMM$

$MMM \rightarrow (I \otimes L) \circ (MMM \otimes I) \circ (I \times (I \otimes L))$

Benchmark: Viterbi Decoding

Viterbi decoding (8-bit) on 2.66 GHz Core 2 Duo
 performance [Gbutterflies/s]

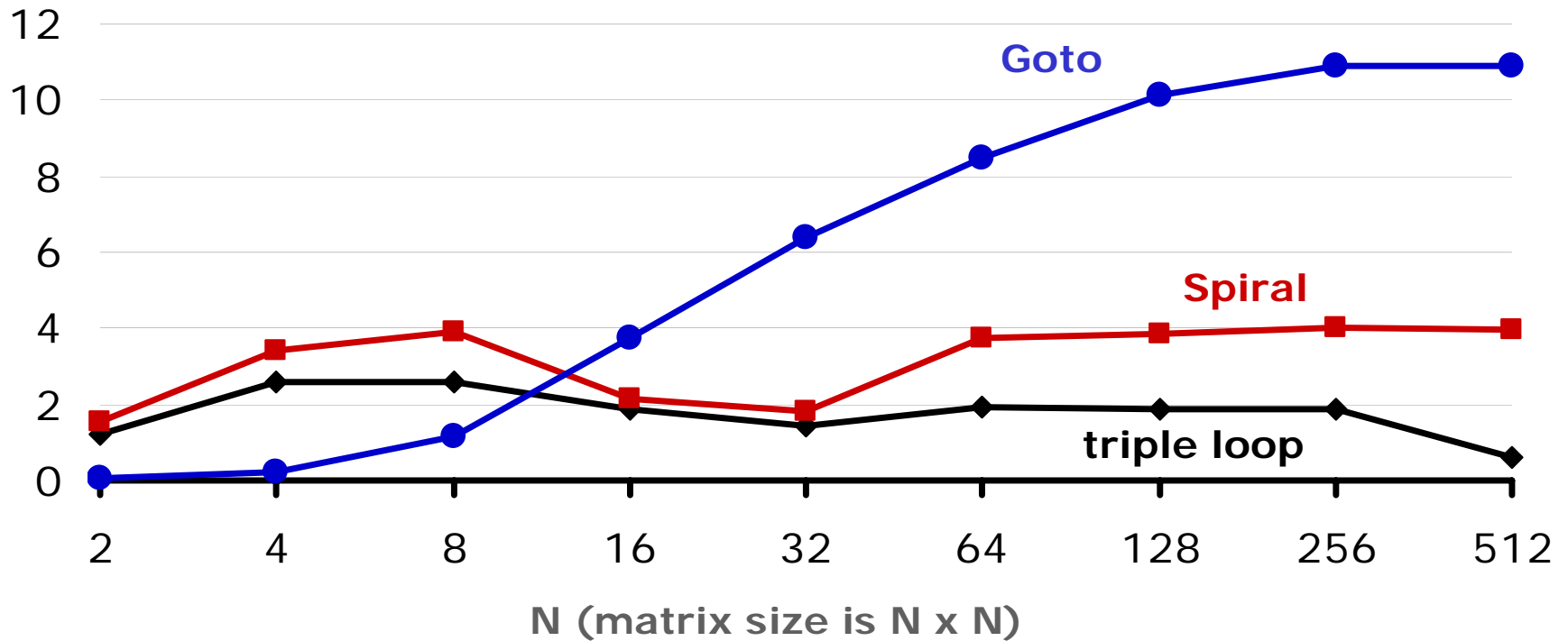
1 butterfly
 = ~22 ops



Vectorized using practically the same rules as for DFT

Benchmark Matrix-Matrix Multiplication

DGEMM on 3 GHz Core 2 Duo (1 thread)
performance [Gflop/s]



Organization

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

Conclusions

- **Spiral: The computer implements and optimizes transforms**
 - From problem specification to very fast code---automatically
 - Retargetable to new platform paradigms (vector, SMP, GPU, FPGA, ...)
 - The generated code is often very fast

- **Key ingredients**
 - Declarative representation of algorithms
 - Optimization at a high 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)