Autotuning (2/2): Specialized code generators

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.18] Thursday, March 6, 2008

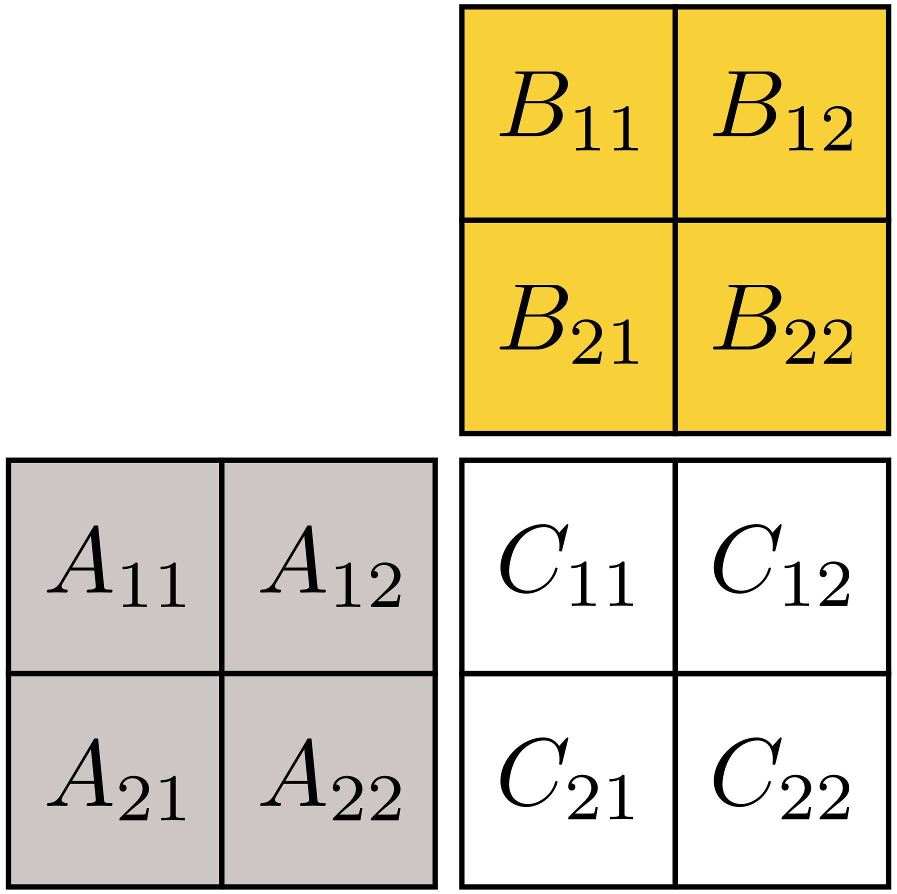
Today’s sources

- CS 267 at UCB (Demmel & Yelick)
- Papers from various autotuning projects
    - PHiPAC, ATLAS, FFTW, SPIRAL, TCE
    - See: Proc. IEEE 2005 special issue on Program Generation, Optimization, and Platform Adaptation
- Me (for once!)

Review:
Cache-oblivious algorithms

A recursive algorithm for matrix-multiply

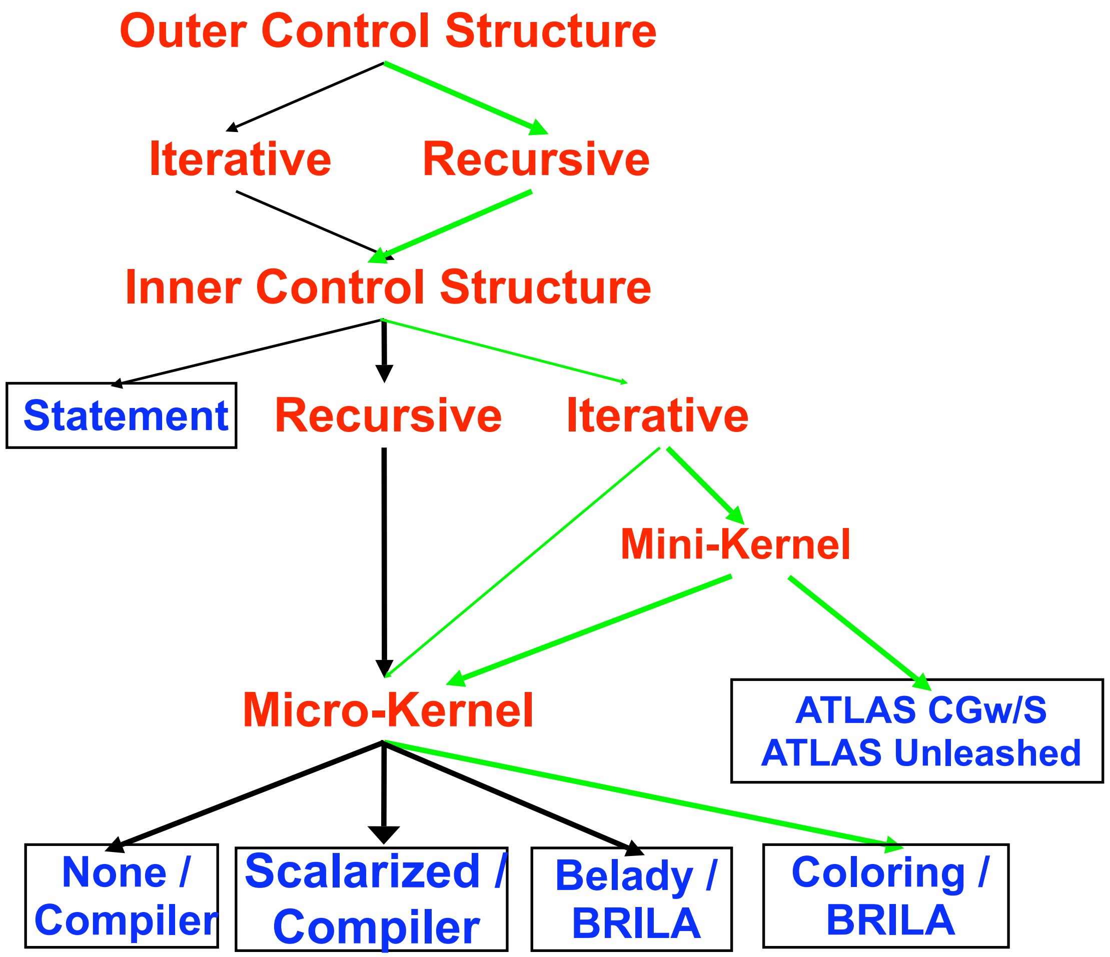
- Divide all dimensions in half
- Bilardi, *et al*.: Use grey-code ordering



No. of misses, with tall-cache assumption:

$$Q(n) = \left\{ \begin{array}{ll} 8 \cdot Q(\frac{n}{2}) & \text{if } n > \sqrt{\frac{M}{3}} \\ 3n^2 & \text{otherwise} \end{array} \right\} \leq \Theta\left(\frac{n^3}{L\sqrt{M}}\right)$$

Performance-engineering challenges



Outer Control Structure

Iterative

Recursive

Inner Control Structure

Statement

Recursive

Iterative

Mini-Kernel

Micro-Kernel

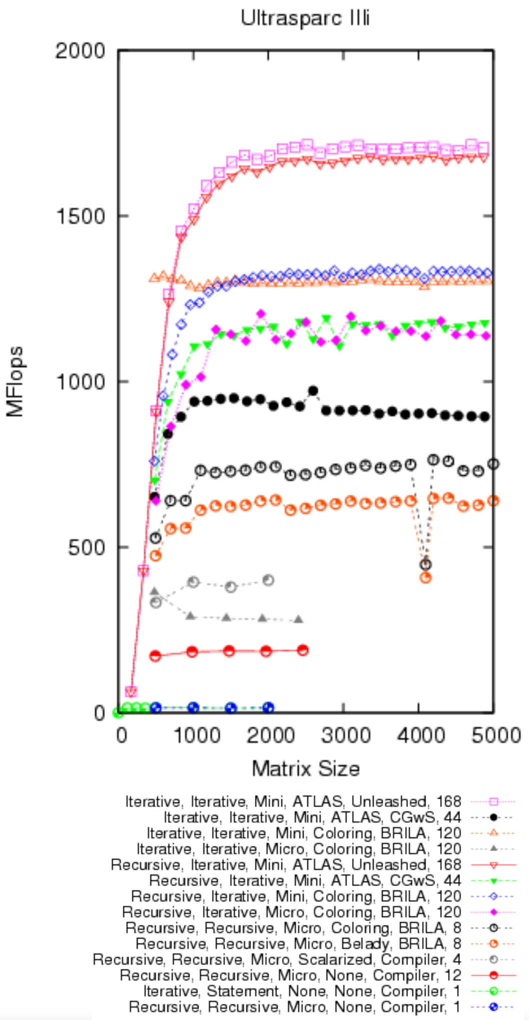
ATLAS CGw/S
ATLAS Unleashed

None / Compiler

Scalarized / Compiler

Belady / BRILA

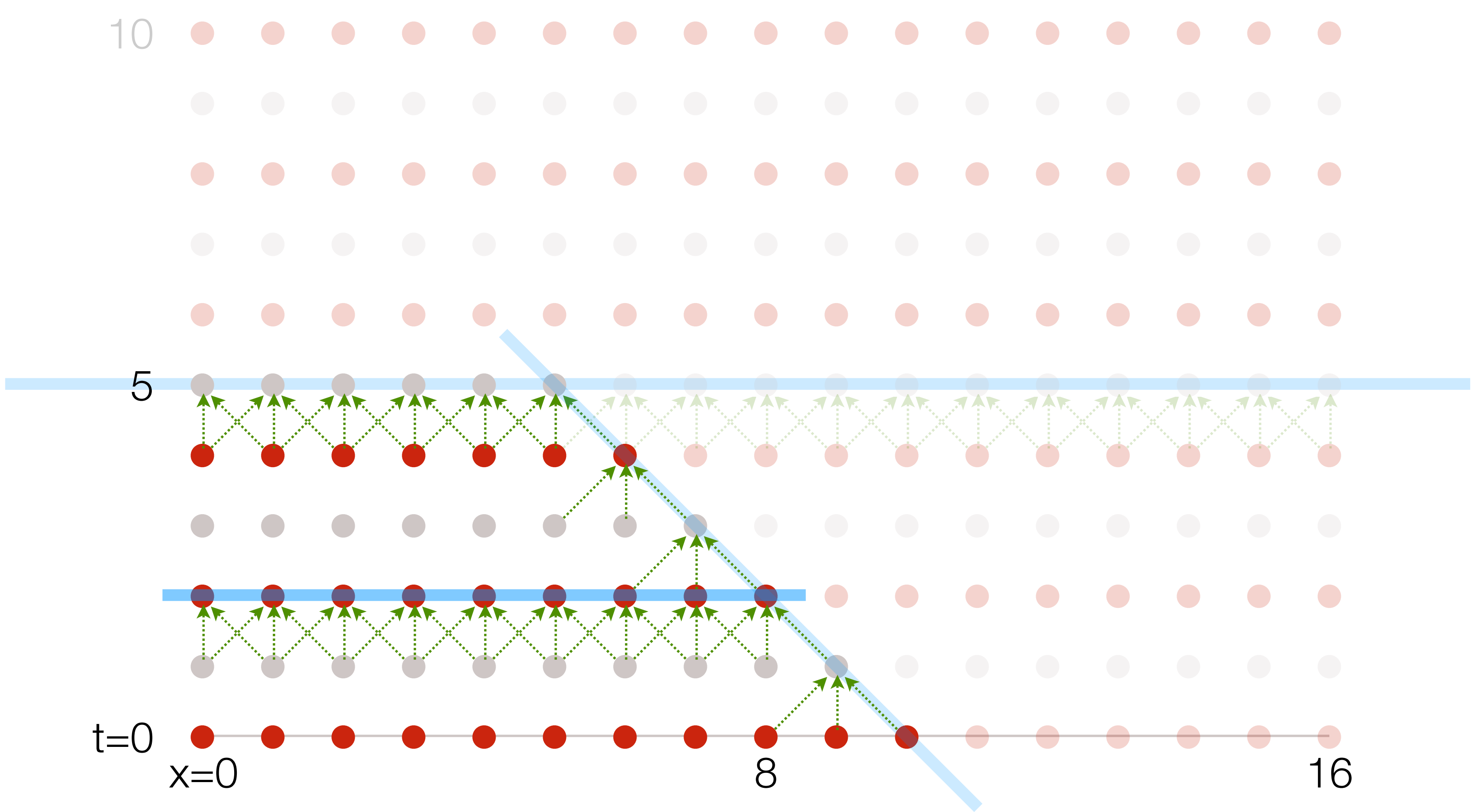
Coloring / BRILA



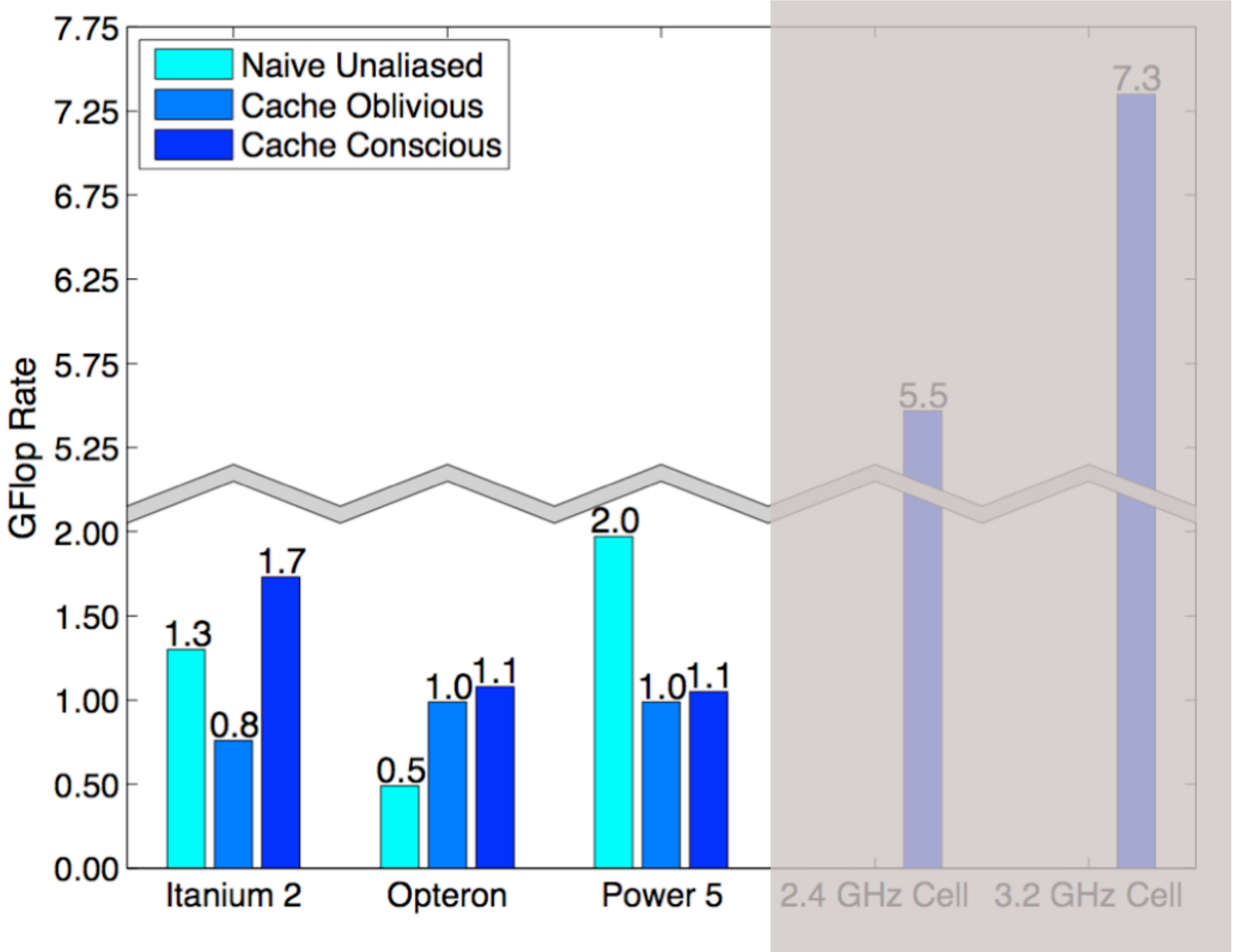
Cache-oblivious stencil computation

Theorem [Frigo & Strumpen (ICS 2005)]:
***d* = dimension ⇒**

$$Q(n, t; d) \quad = \quad O\left(\frac{n^d \cdot t}{M^{\frac{1}{d}}}\right)$$



Cache-conscious algorithm



Source: Datta, et al. *(2007)*

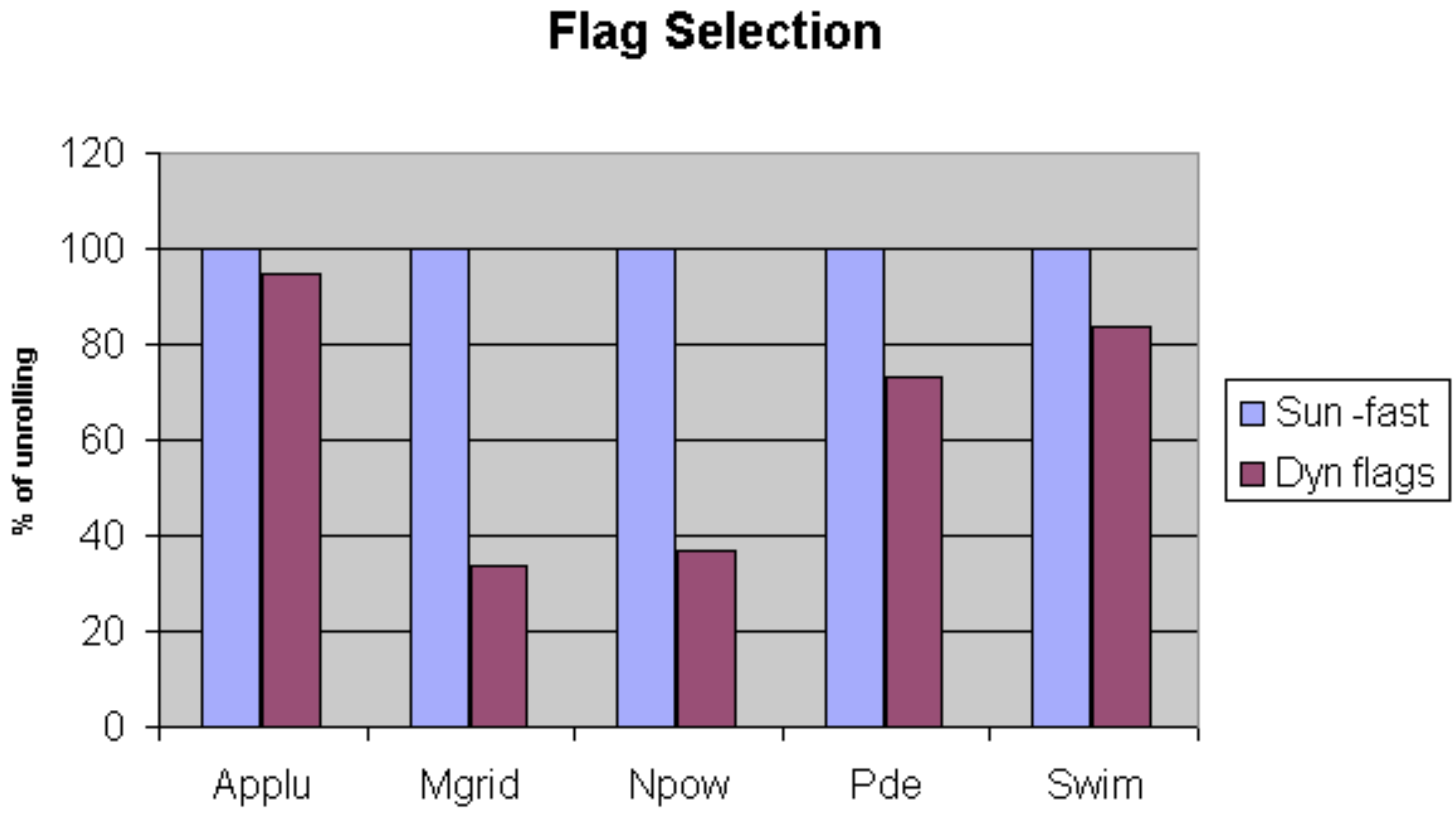
Survey of autotuning

Early idea seedlings

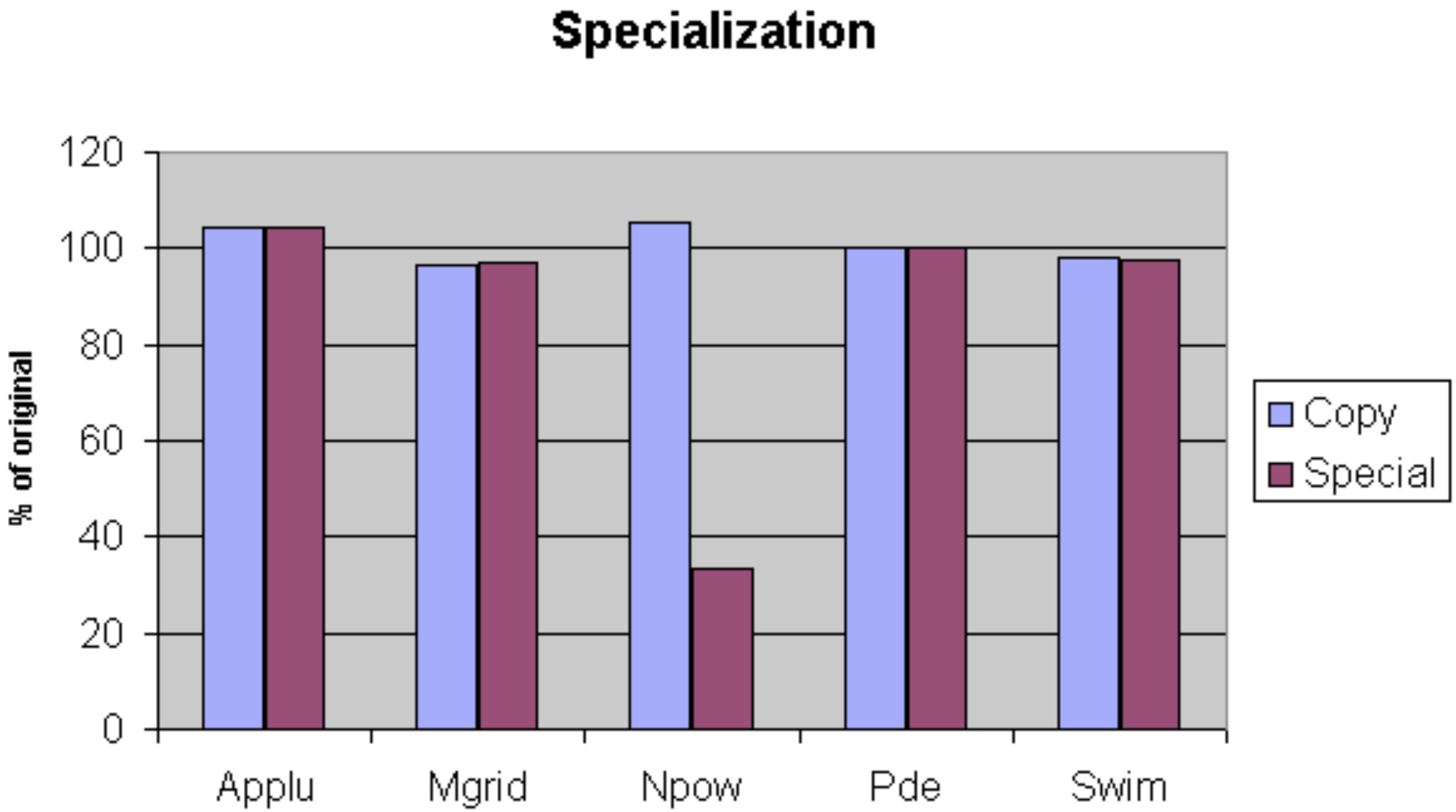
- **Polyalgorithms**: John R. Rice
  - (1969) “A polyalgorithm for the automatic solution of nonlinear equations”
  - (1976) “The algorithm selection problem”
- **Profiling** and feedback-directed compilation
  - (1971) D. Knuth: “An empirical study of FORTRAN programs”
  - (1982) S. Graham, P. Kessler, M. McKusick: gprof
  - (1991) P. Chang, S. Mahlke, W-m. W. Hwu: “Using profile information to assist classic code optimizations”
- Code generation from **high-level representations**
  - (1989) J. Johnson, R.W. Johnson, D. Rodriguez, R. Tolimieri: “A methodology for designing, modifying, and implementing Fourier Transform algorithms on various architectures.”
  - (1992) M. Covell, C. Myers, A. Oppenheim: “Computer-aided algorithm design and arrangement” (1992)

Why doesn’t the compiler do the dirty work?

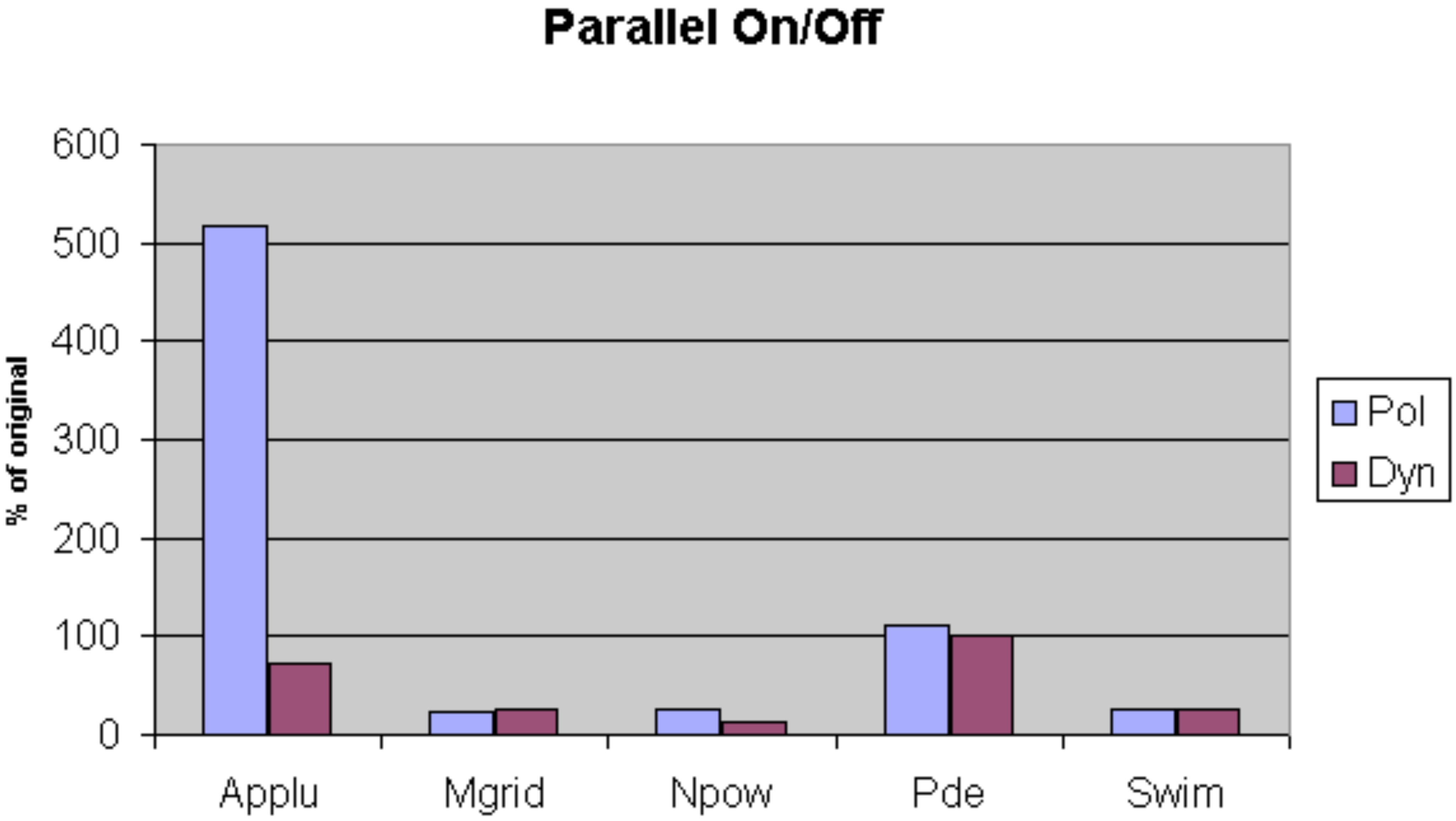
- Why doesn’t the compiler do all of this?
  - Analysis
    - Over-specified dependencies
    - Correctness requirements
    - Limited access to relevant run-time information
  - Architecture: Realistic hardware models?
  - Engineering: Hard to modify a production compiler



Source: Voss, ADAPT compiler project: http://www.eecg.toronto.edu/~voss/AdaptPage/results.html



Source: Voss, ADAPT compiler project: http://www.eecg.toronto.edu/~voss/AdaptPage/results.html



Source: Voss, ADAPT compiler project: http://www.eecg.toronto.edu/~voss/AdaptPage/results.html

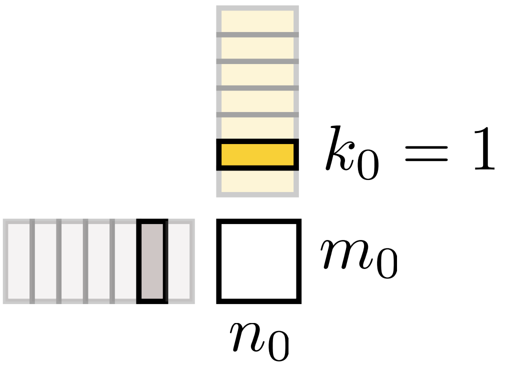
Automatic performance tuning, or "autotuning"

- Two-phase methodology for producing automatically tuned code
  - Given: Computational kernel or program; inputs; machine
  - Identify and generate a parameterized space of candidate implementations
  - Select the fastest one using empirical modeling and automated experiments
- "Autotuner" = System that implements this
  - Usually domain-specific (exception: "autotuning/iterative compilers")
  - Leverage back-end compiler for performance and portability

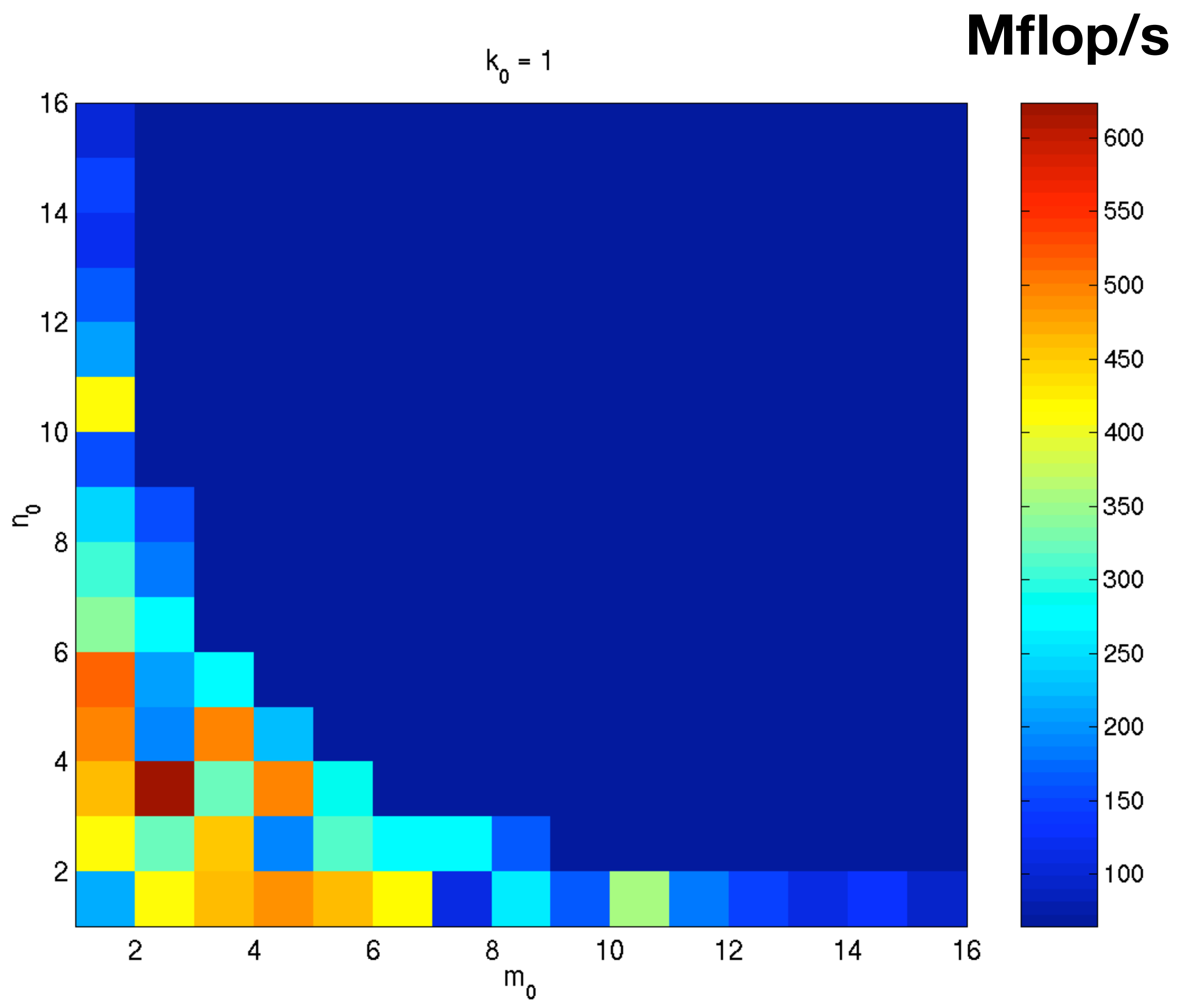
How an autotuner differs from a compiler (roughly)

	Compiler	Autotuner
Input	General-purpose source code	Specification
Code generation time	User responsive	Long, but amortized
Implementation selection	Static analysis; some run-time profiling/feedback	Automated empirical models and experiments

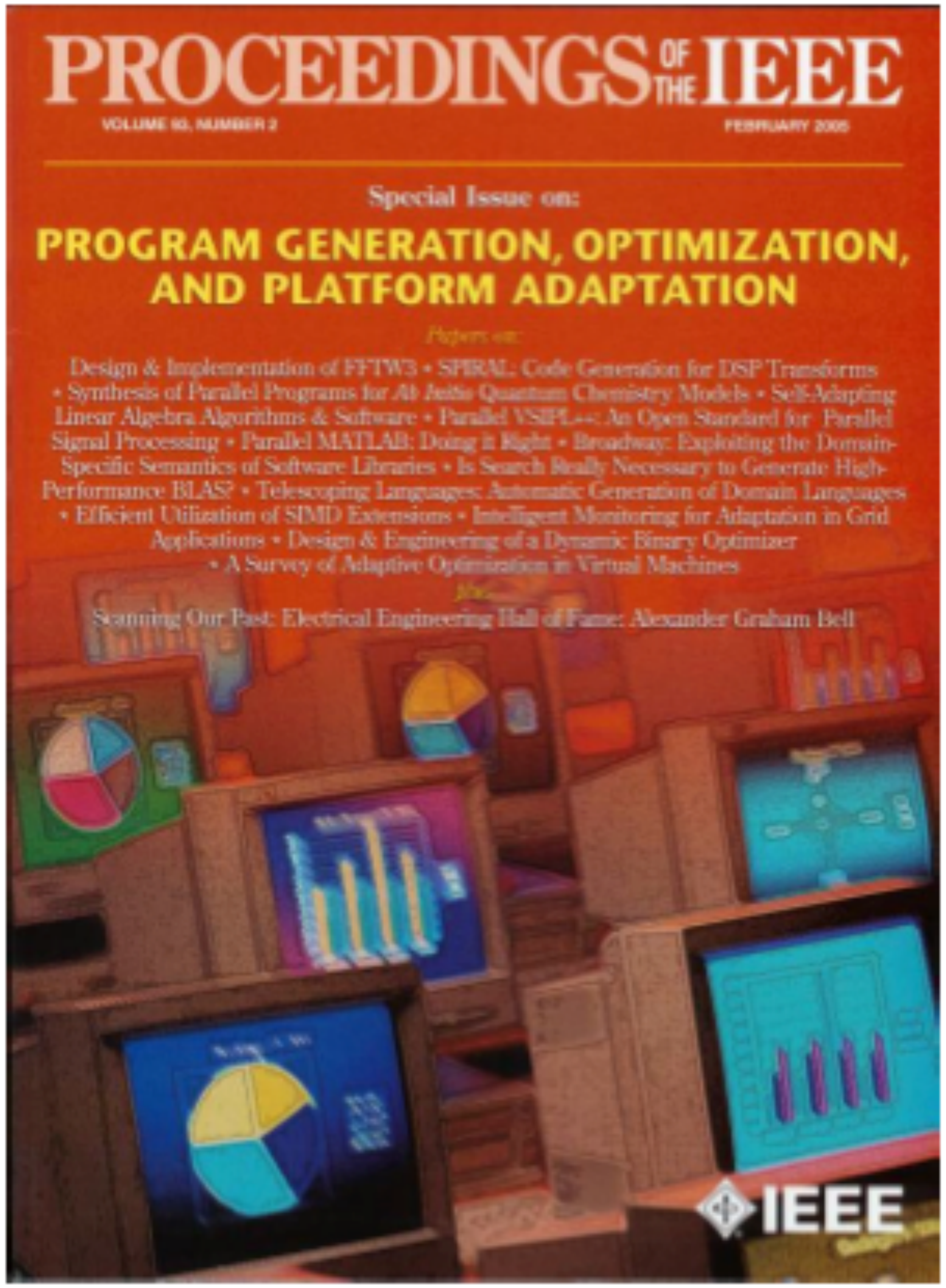
Example: What a search space looks like



- Platform: Sun Ultra IIi
  - 16 double regs
  - 667 Mflop/s peak
  - Unrolled, pipelined inner-kernel
  - Sun cc v5.0 compiler



Source: PHiPAC Project at UC Berkeley (1997)



PROCEEDINGS OF THE IEEE

VOLUME 93, NUMBER 2

FEBRUARY 2005

Special Issue on:

PROGRAM GENERATION, OPTIMIZATION, AND PLATFORM ADAPTATION

Papers on:

Design & Implementation of FFTW3 • SPIRAL: Code Generation for DSP Transforms • Synthesis of Parallel Programs for *Ab Initio* Quantum Chemistry Models • Self-Adapting Linear Algebra Algorithms & Software • Parallel VSIPL++: An Open Standard for Parallel Signal Processing • Parallel MATLAB: Doing it Right • Broadway: Exploiting the Domain-Specific Semantics of Software Libraries • Is Search Really Necessary to Generate High-Performance BLAS? • Telescoping Languages: Automatic Generation of Domain Languages • Efficient Utilization of SIMD Extensions • Intelligent Monitoring for Adaptation in Grid Applications • Design & Engineering of a Dynamic Binary Optimizer • A Survey of Adaptive Optimization in Virtual Machines

Plus:

Scanning Our Past: Electrical Engineering Hall of Fame: Alexander Graham Bell

Proceedings of the IEEE special issue, Feb. 2005

Dense linear algebra

PHiPAC (1997)

- Portable High-Performance ANSI C [Bilmes, Asanovic, Chin, Demmel (1997)]
  - Coding guidelines: C as high-level assembly language
  - Code generator for multi-level cache- and register-blocked matrix multiply
  - Exhaustive search over all parameters
  - Began as class project which beat the vendor BLAS

PHiPAC coding guideline example: Removing false dependencies

- Use local variables to remove false dependencies

```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d;
```

False read-after-write hazard between a[i] and b[i+1]

↓

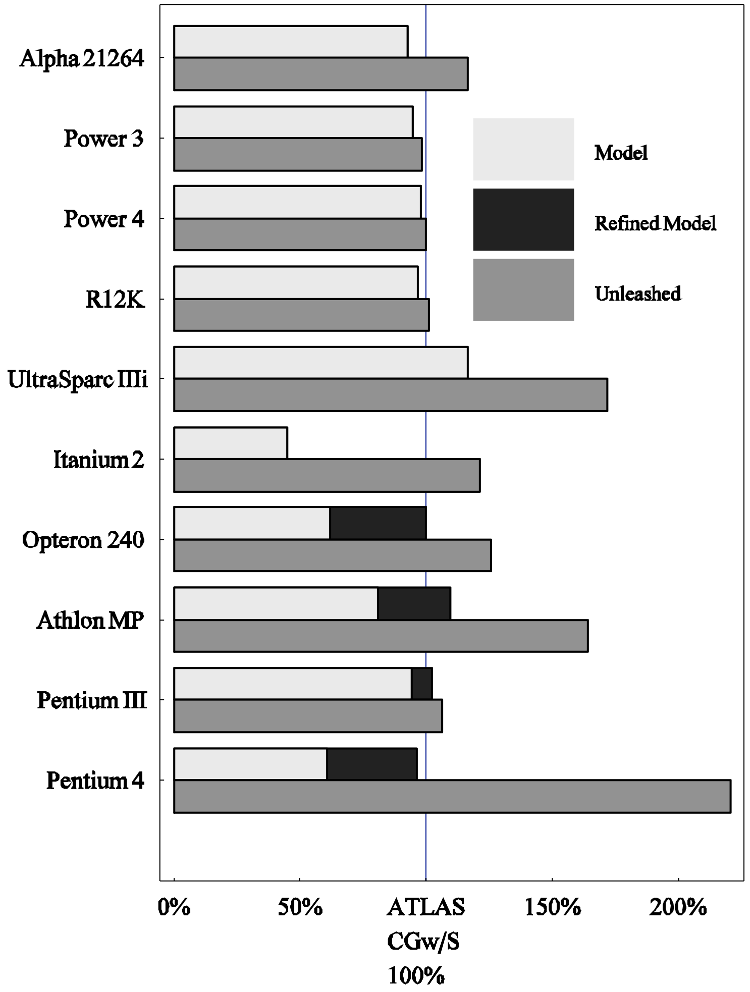
```
float f1 = b[i];
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 * d;
```

In C99, may declare a & b unaliased (“restrict” keyword)

ATLAS (1998)

- “Automatically Tuned Linear Algebra Software” — [R.C. Whaley and J. Dongarra (1998)]
  - Overcame PHiPAC shortcomings on x86 platforms
  - Copy optimization, prefetch, alternative schedulings
  - Extended to full BLAS, some LAPACK support (*e.g.*, LU)
- Code generator (written in C, output C w/ inline-assembly) with search
  - Copy optimization prunes much of PHiPAC’s search space
  - “Simple” line searches
  - See: iterative floating-point kernel optimizer (iFKO) work

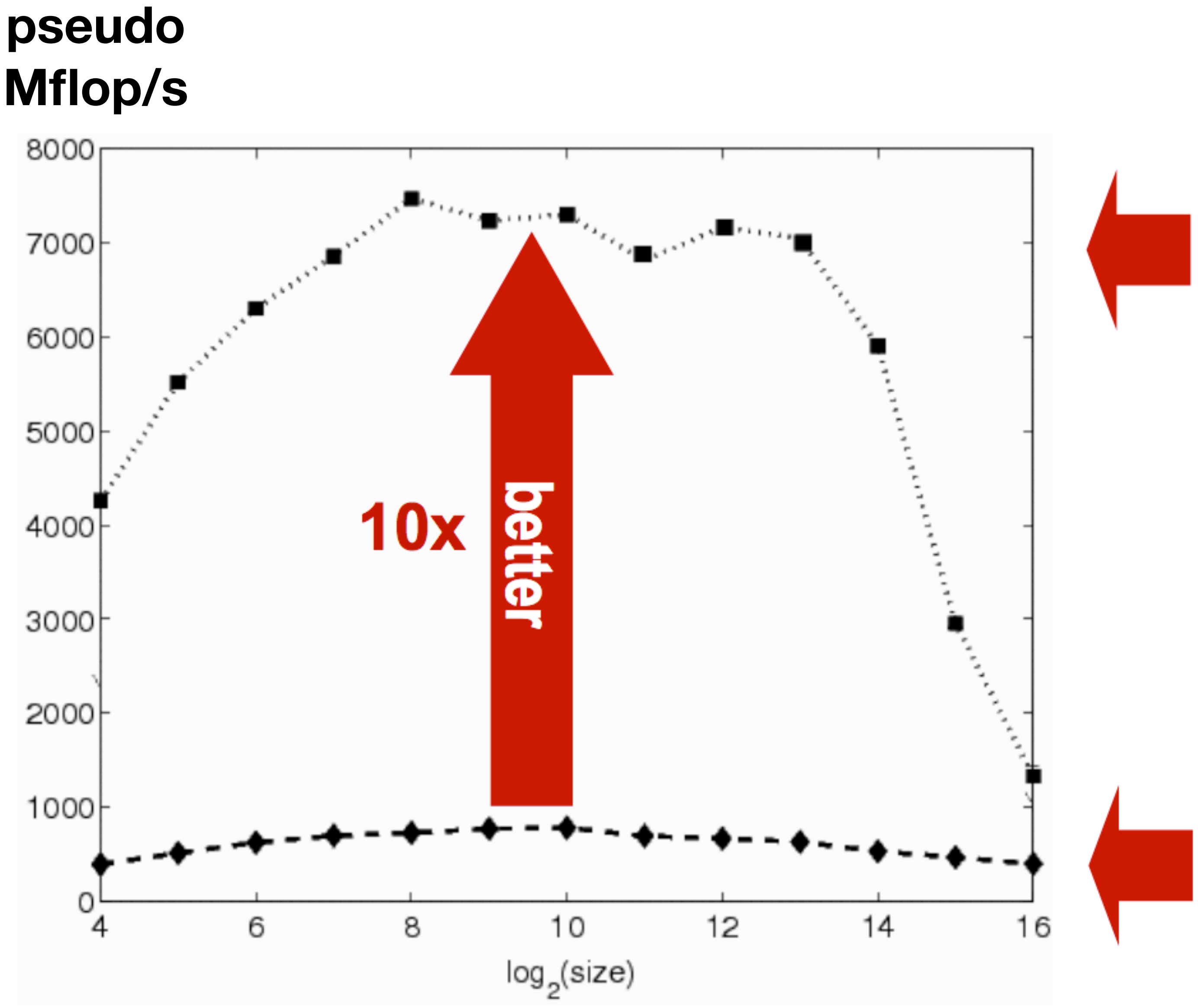
Search vs. modeling



- Yotov, *et al*. “Is search really necessary to generate high-performance BLAS?”
- “Think globally, search locally”
    - Small gaps ⇒ local search
    - Large gaps ⇒ refine model
- “Unleashed” ⇒ hand-optimized plug-in kernels

Signal processing

Motivation for performance tuning



best available implementation (FFTW, Intel IPP, Spiral)

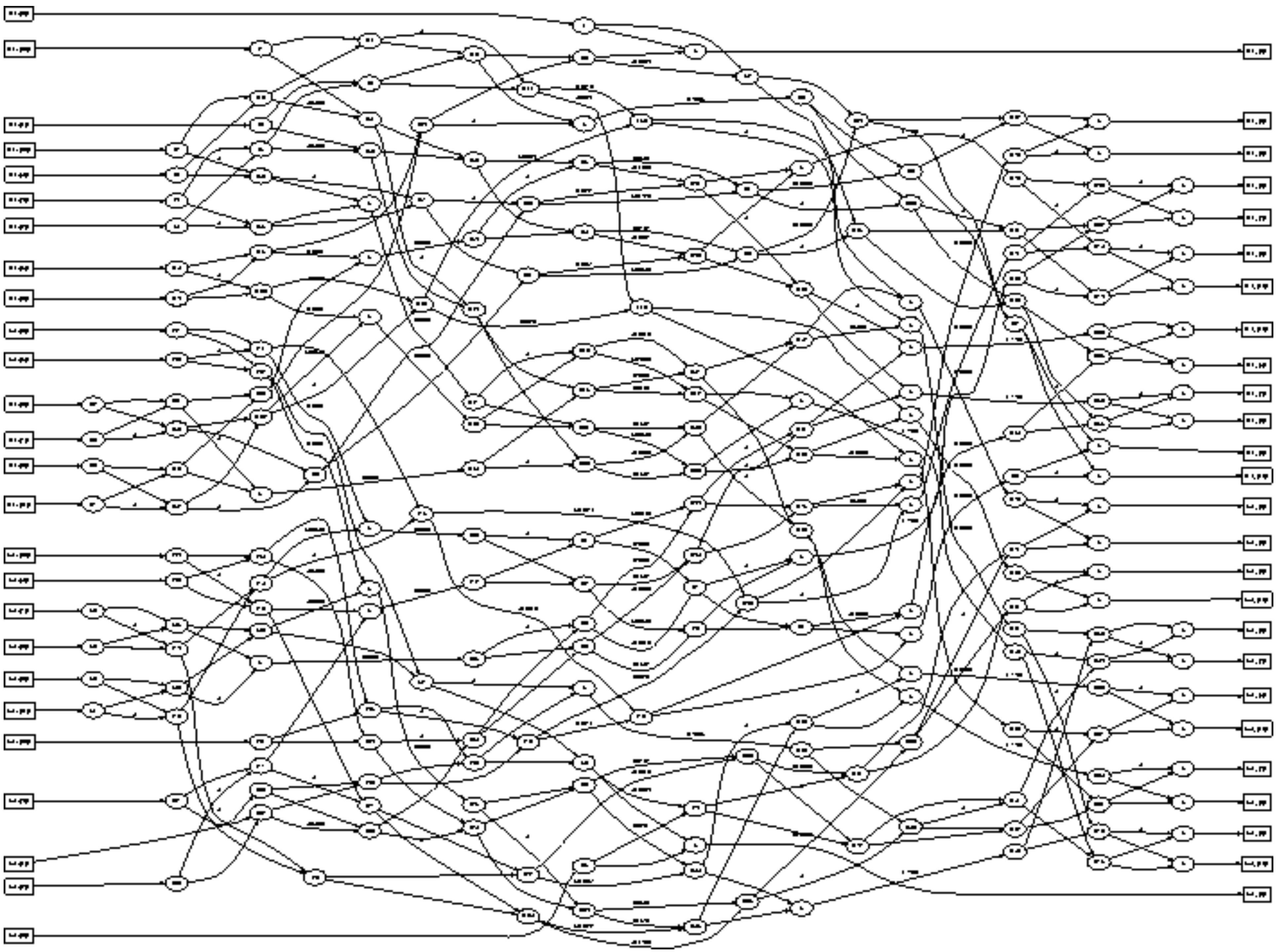
roughly the same operations count

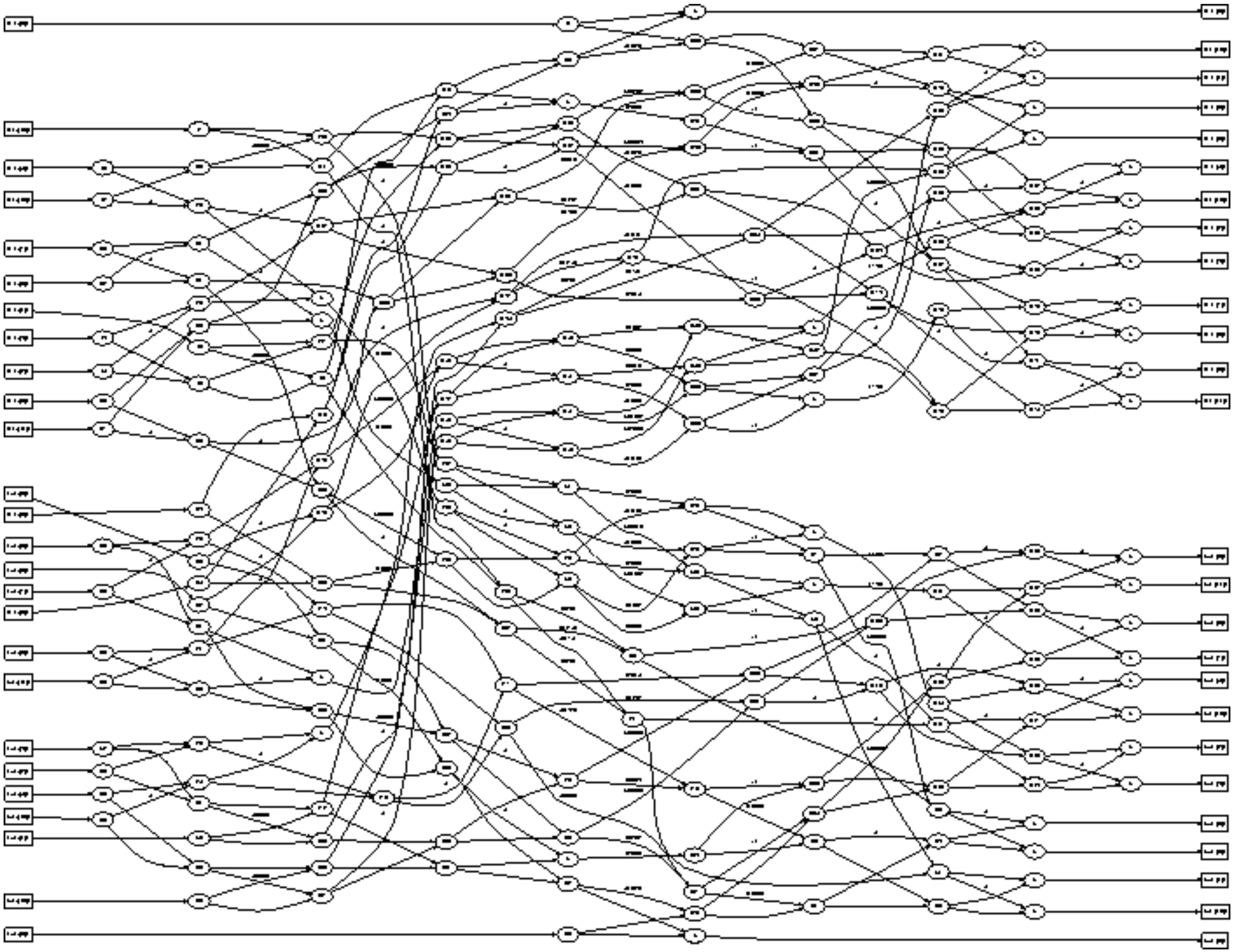
reasonable implementation (Numerical recipes, GNU scientific library)

Source: J. Johnson (2007), CScADS autotuning workshop

FFTW (1997)

- “Fastest Fourier Transform in the West” [M. Frigo, S. Johnson (1997)]
- “**Codelet**” generator (in OCaml)
    - Explicit represent a small fixed-size transform by its computation DAG
    - Optimize DAG: Algebraic transformations, constant folding, “DAG transposition”
    - Schedule DAG cache-obliviously and output as C source code
- **Planner**: At run-time, determine which codelets to apply
- **Executor**: Perform FFT of a particular size using plan
- Efficient “plug-in” assembly kernels





Cooley-Tukey FFT algorithm

$$\begin{aligned} y[k] &\quad\leftarrow\quad \mathrm{DFT}_N(x,k) \equiv \sum_{j=0}^{N-1} x[j]\cdot\omega_N^{-kj} \qquad\qquad x,y\in\mathbb{C}^N \\ \omega_N &\quad\equiv\quad e^{2\pi\sqrt{-1}/N} \\ N &\quad\equiv\quad N_1\cdot N_2 \end{aligned}$$

Cooley-Tukey FFT algorithm

$$
\begin{array}{rcl}
y[k] & \leftarrow & \mathrm{DFT}_N(x,k) \equiv \sum_{j=0}^{N-1} x[j]\cdot\omega_N^{-kj} \qquad\qquad x, y \in \mathbb{C}^N \\
\omega_N & \equiv & e^{2\pi\sqrt{-1}/N} \\
N & \equiv & N_1 \cdot N_2 \\
 & \Downarrow & \\
0 \le k_1 < N_1 & \text{and} & 0 \le k_2 < N_2 \\
y[k_1 + k_2 \cdot N_1] & \leftarrow & \underbrace{\sum_{n_2=0}^{N_2-1} \left[\underbrace{\left(\sum_{n_1}^{N_1-1} x[n_1 \cdot N_2 + n_2] \cdot \omega_{N_1}^{-k_1 n_1} \right)}_{N_1\text{-point DFT}} \underbrace{\cdot\,\omega_N^{-k_1 n_2}}_{\text{Twiddle}} \right] \cdot\,\omega_{N_2}^{-k_2 n_2}}_{N_2\text{-point DFT}}
\end{array}
$$

Cooley-Tukey FFT algorithm: Encoding in the codelet generator

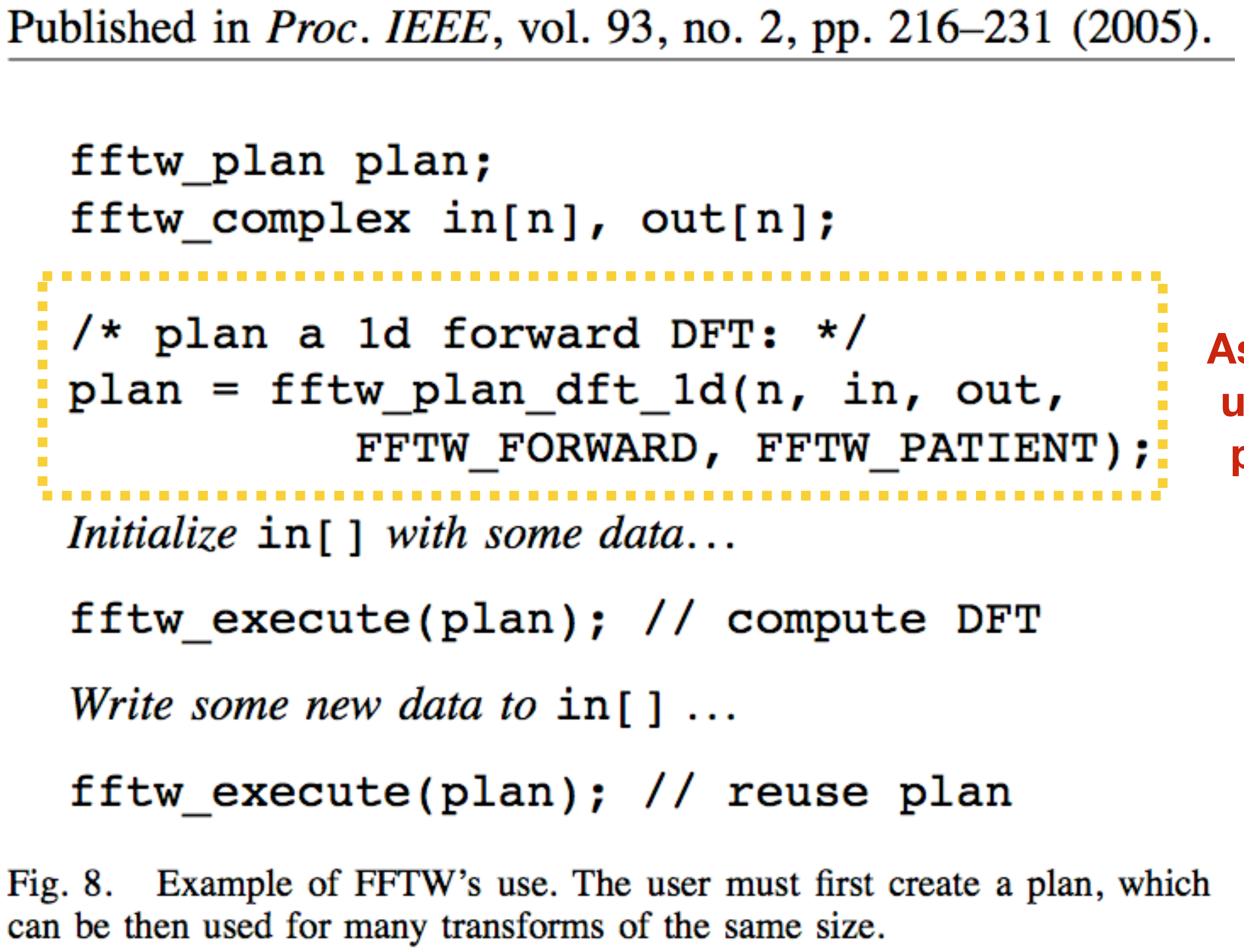
$$y[k] \quad \leftarrow \quad \mathrm{DFT}_N(x,k) \;\equiv\; \sum_{j=0}^{N-1} x[j]\cdot\omega_N^{-kj} \qquad\qquad x, y \in \mathbb{C}^N$$

$$y[k_1 + k_2\cdot N_1] \quad \leftarrow \quad \underbrace{\sum_{n_2=0}^{N_2-1}\left[\underbrace{\left(\sum_{n_1}^{N_1-1} x[n_1\cdot N_2 + n_2]\cdot\omega_{N_1}^{-k_1 n_1}\right)}_{N_1\text{-point DFT}} \underbrace{\cdot\,\omega_N^{-k_1 n_2}}_{\text{Twiddle}}\right]\cdot\omega_{N_2}^{-k_2 n_2}}_{N_2\text{-point DFT}}$$

```
let dftgen$(N, x) \equiv$ fun $k \rightarrow \ldots$    # $\mathrm{DFT}_N(x,k)$
let cooley_tukey$(N_1, N_2, x) \equiv$
    let $\hat{x} \equiv$ fun $n_2, n_1 \rightarrow x(n_2 + n_1\cdot N_2)$ in
    let $\mathbf{G_1} \equiv$ fun $n_2 \rightarrow$ dftgen$(N_1, \hat{x}(n_2, \_))$ in
    let $\mathbf{W} \equiv$ fun $k_1, n_2 \rightarrow \mathbf{G_1}(n_2, k_1)\cdot\omega_N^{-k_1 n_2}$ in
    let $\mathbf{G_2} \equiv$ fun $k_1 \rightarrow$ dftgen$(N_2, \mathbf{W}(k_1, \_))$
    in
      fun $k \rightarrow \mathbf{G_2}(k \bmod N_1, k \operatorname{div} N_1)$
```

(Functional pseudo-code)

Planner phase



Published in *Proc. IEEE*, vol. 93, no. 2, pp. 216–231 (2005).

```
fftw_plan plan;
fftw_complex in[n], out[n];

/* plan a 1d forward DFT: */
plan = fftw_plan_dft_1d(n, in, out,
                        FFTW_FORWARD, FFTW_PATIENT);
```

Initialize `in[ ]` *with some data...*

```
fftw_execute(plan); // compute DFT
```

Write some new data to `in[ ]` ...

```
fftw_execute(plan); // reuse plan
```

Fig. 8. Example of FFTW's use. The user must first create a plan, which can be then used for many transforms of the same size.

Assembles plan using dynamic programming

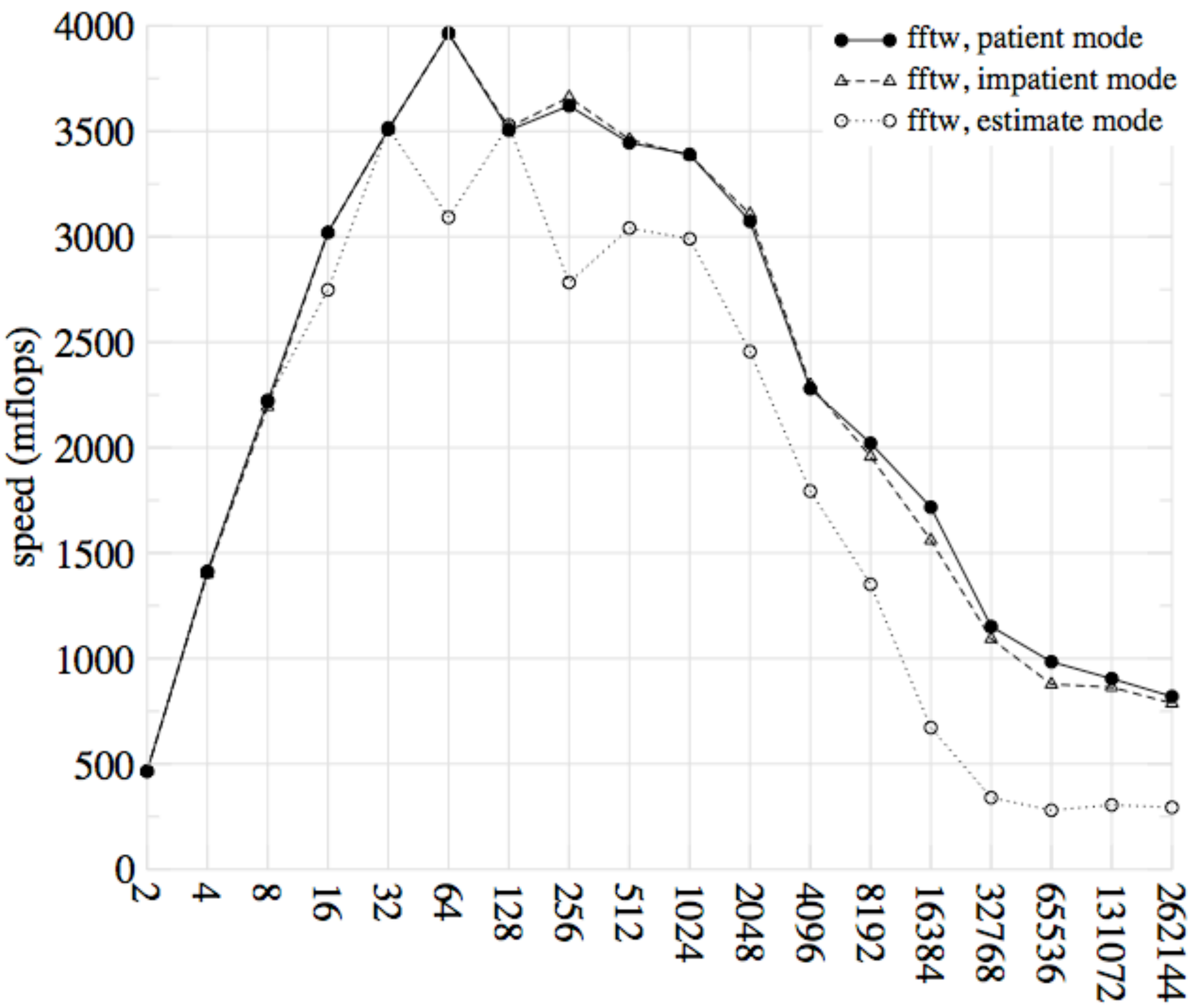


Fig. 9. Effect of planner tradeoffs: comparison of patient, impatient, and estimate modes in FFTW for double-precision 1d complex DFTs, power-of-two sizes, on a 2 GHz PowerPC 970 (G5). Compiler and flags as in Fig. 4.

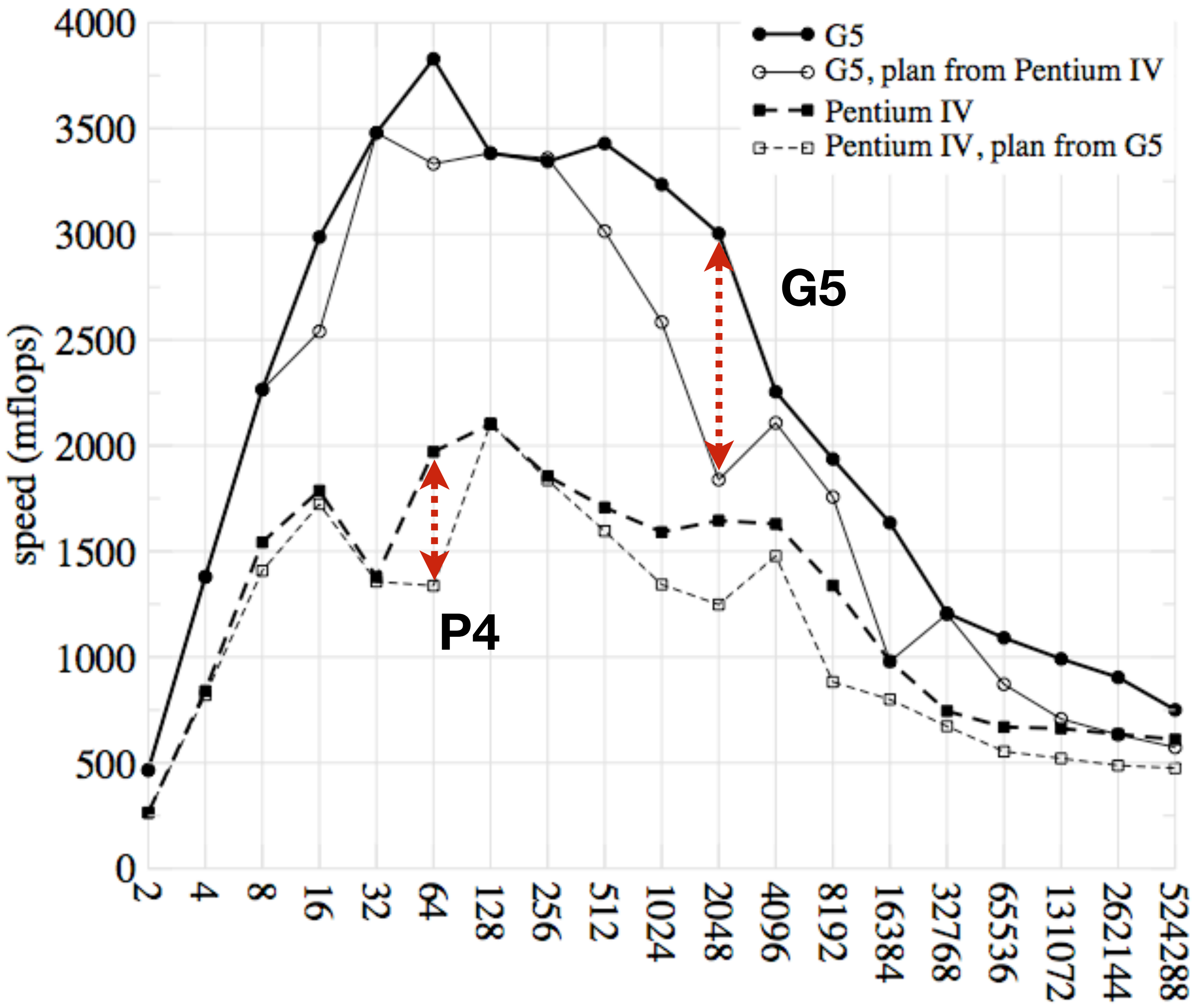


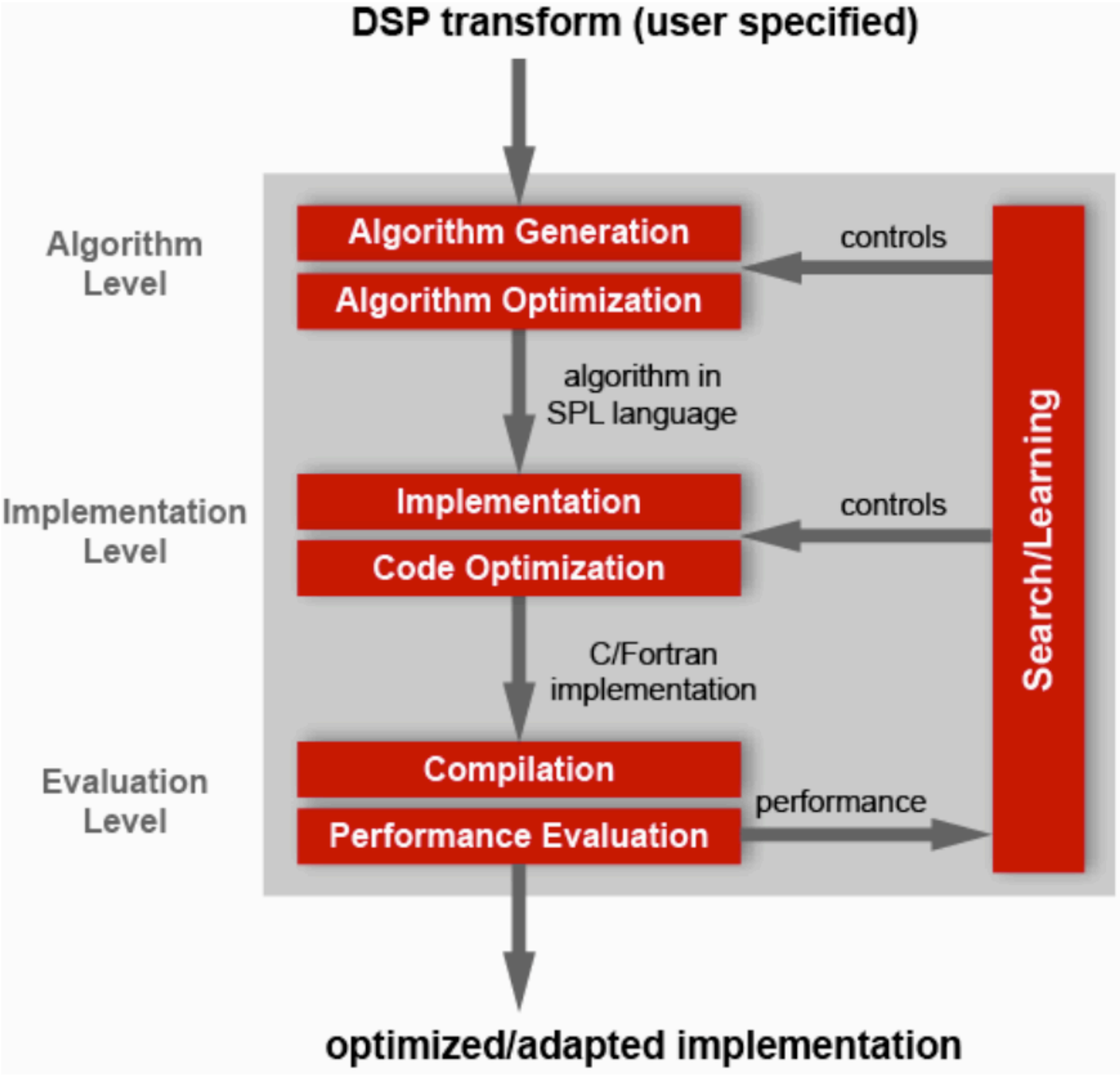
Fig. 10. Effects of tuning FFTW on one machine and running it on another. The graph shows the performance of one-dimensional DFTs on two machines: a 2 GHz PowerPC 970 (G5), and a 2.8 GHz Pentium IV. For each machine, we report both the speed of FFTW tuned to that machine and the speed tuned to the *other* machine.

SPIRAL

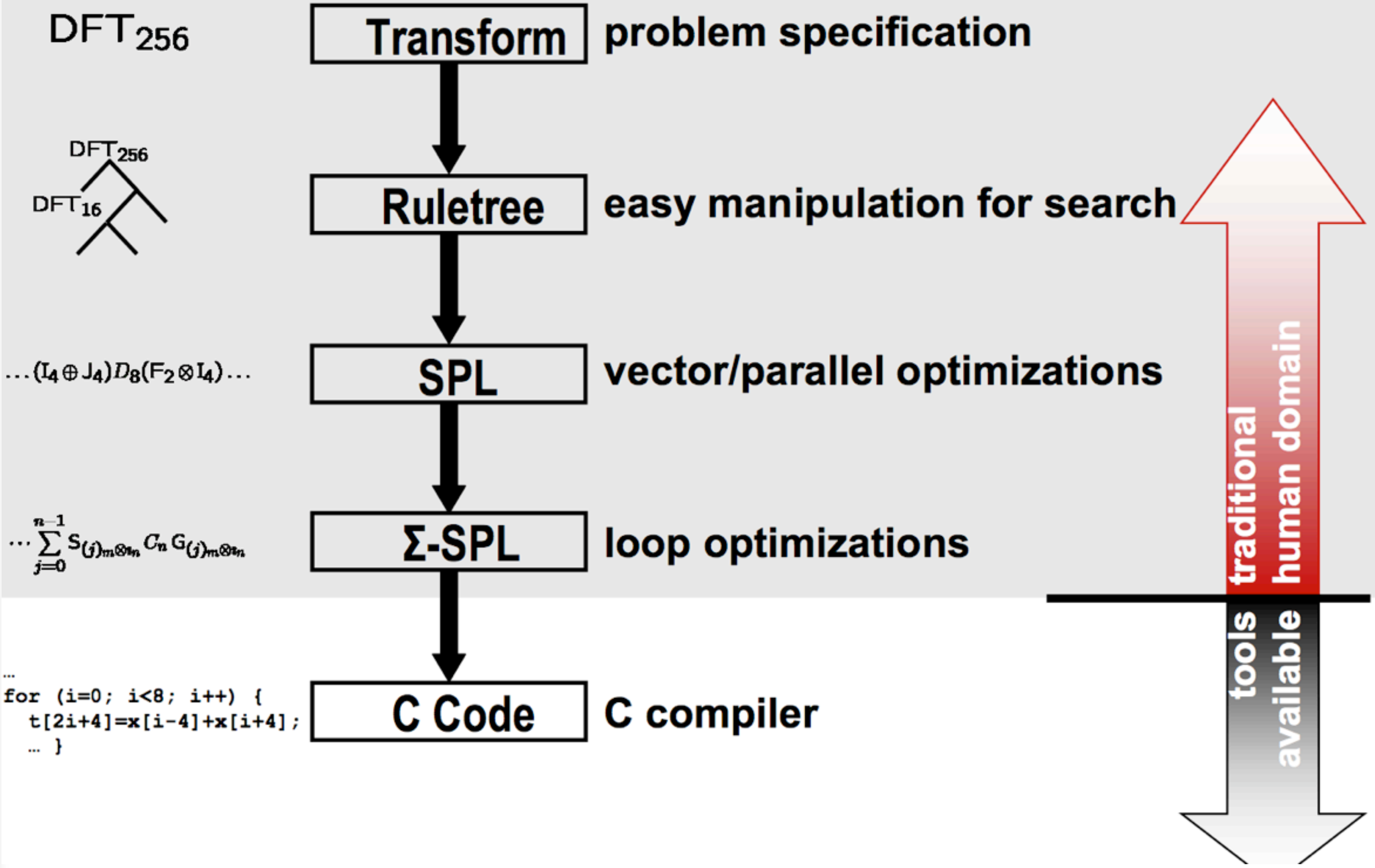
Software/Hardware Generation for DSP Algorithms

SPIRAL (1998)

- Code generator
  - Represent linear transformations as formulas
  - Symbolic algebra + rewrite engine transforms formulas
- Search using variety of techniques (more later)



Source: J. Johnson (2007), CScADS autotuning workshop



DFT_{256} **Transform** problem specification

Ruletree easy manipulation for search

$\ldots(I_4 \oplus J_4) D_8 (F_2 \otimes I_4)\ldots$ **SPL** vector/parallel optimizations

$\ldots\sum_{j=0}^{n-1} S_{(j)_{m\otimes n}} C_n G_{(j)_{m\otimes n}}$ **Σ-SPL** loop optimizations

```
...
for (i=0; i<8; i++) {
  t[2i+4]=x[i-4]+x[i+4];
  ... }
```

C Code C compiler

Source: J. Johnson (2007), CScADS autotuning workshop

High-level representations and rewrite rules

$$\begin{aligned}
\mathbf{DFT}_N &\equiv \left[\omega_N^{kl}\right]_{0\le k,l<N} \\
\mathbf{DCT\text{-}2}_N &\equiv \left[\cos\frac{(2l+1)k\pi}{2N}\right]_{0\le k,l<N}
\end{aligned}$$

$$\vdots$$

$$\begin{aligned}
& n = k\cdot m : \\
\Longrightarrow\ \mathbf{DFT}_n &\rightarrow (\mathbf{DFT}_k \otimes I_m) T_m^n (I_k \otimes \mathbf{DFT}_m) L_k^n \\
& n = k\cdot m,\ \gcd(k,m)=1 : \\
\Longrightarrow\ \mathbf{DFT}_n &\rightarrow P_n(\mathbf{DFT}_k \otimes \mathbf{DFT}_m) Q_n \\
& p \text{ is prime} : \\
\Longrightarrow\ \mathbf{DFT}_p &\rightarrow R_p^T (I_1 \oplus \mathbf{DFT}_{p-1} D_p (I_1 \oplus \mathbf{DFT}_{p-1}) R_p
\end{aligned}$$

$$\vdots$$

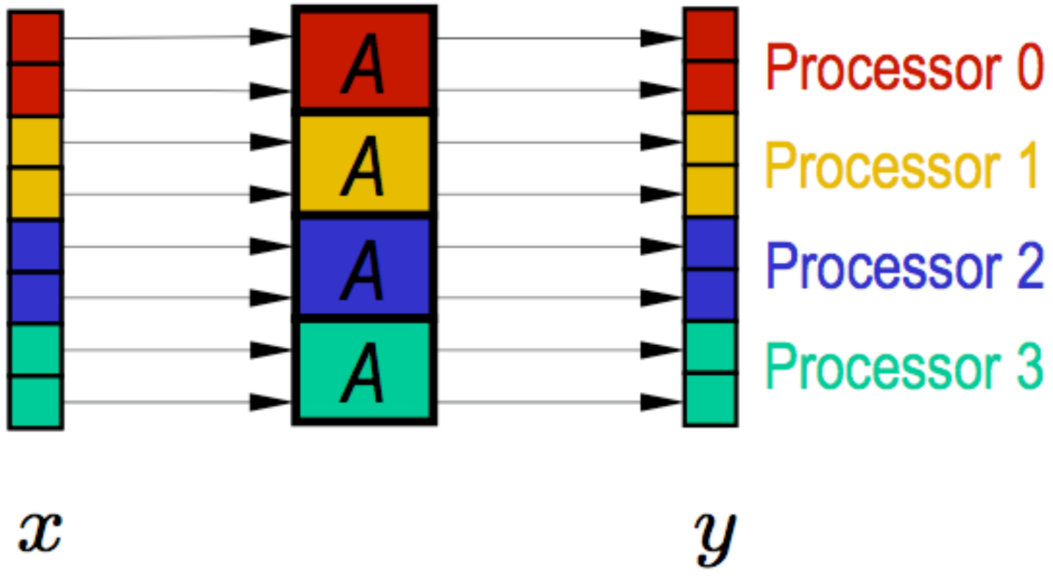
$$\mathbf{DFT}_2 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

High-level representations expose parallelism

$$
(I_4 \otimes A) \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} = \begin{bmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}
$$

$$
= \begin{bmatrix} AX_1 \\ AX_2 \\ AX_3 \\ AX_4 \end{bmatrix}
$$

***A* applied 4 times independently**



High-level representations expose parallelism

$$\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_2\right) \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} a \cdot I_2 & b \cdot I_2 \\ c \cdot I_2 & d \cdot I_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

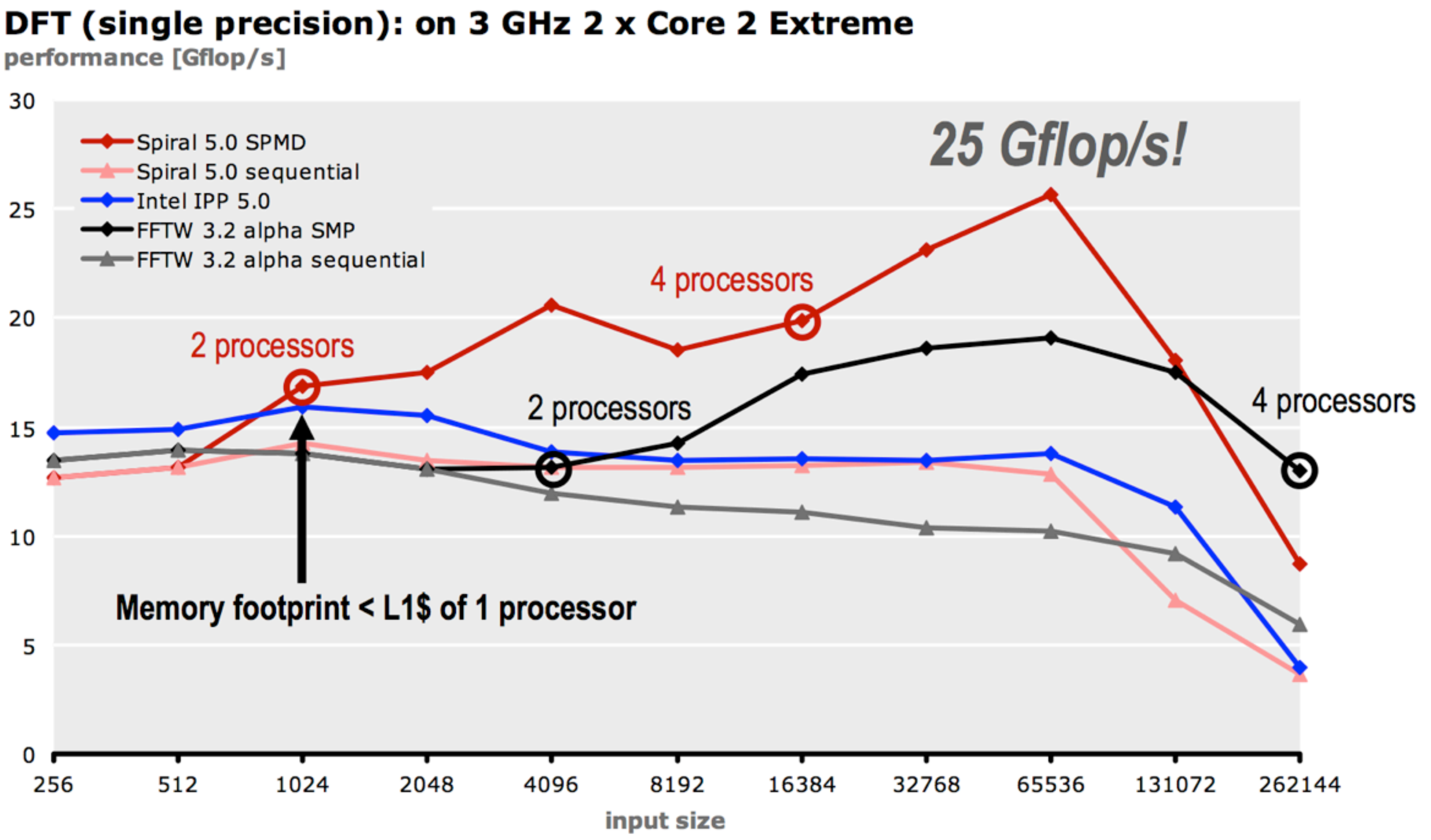
$$= \begin{bmatrix} a \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} \\ c \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + d \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} \end{bmatrix}$$

SIMD-vectorizable

Search in SPIRAL

- Search over ruletrees, i.e., possible formula expansions
- Empirical search
  - Exhaustive
  - Random
  - Dynamic programming
  - Evolutionary search
  - Hill climbing
  - Machine learning methods

Example: SMP + vectorization results



4-way vectorized + up to 4-threaded + adapted to the memory hierarchy

Source: F. Franchetti (2007), CScADS autotuning workshop

Administrivia

Upcoming schedule changes

- Some adjustment of topics (TBD)
- Tu 3/11 — Project proposals due
- Th 3/13 — SIAM Parallel Processing (attendance encouraged)
- Tu 4/1 — No class
- Th 4/3 — Attend talk by Doug Post from DoD HPC Modernization Program

Homework 1:
Parallel conjugate gradients

- Put name on write-up!
- Grading: 100 pts max
    - Correct implementation — 50 pts
    - Evaluation — 30 pts
        - Tested on two samples matrices — 5
        - Implemented and tested on stencil — 10
        - “Explained” performance (e.g., per proc, load balance, comp. vs. comm) — 15
    - Performance model — 15 pts
    - Write-up “quality” — 5 pts

Projects

- **Proposals due Tu 3/11**
- Your goal should be to do something useful, interesting, and/or publishable!
  - Something you’re already working on, suitably adapted for this course
  - Faculty-sponsored/mentored
  - Collaborations encouraged

My criteria for “approving” your project

- “Relevant to this course:” Many themes, so think (and “do”) broadly
  - Parallelism and architectures
  - Numerical algorithms
  - Programming models
  - Performance modeling/analysis

General styles of projects

- Theoretical: Prove something hard (high risk)
- Experimental:
    - Parallelize something
    - Take existing parallel program, and improve it using models & experiments
    - Evaluate algorithm, architecture, or programming model

Examples

- *Anything of interest to a faculty member/project outside CoC*
- Parallel sparse triple product ($R*A*R^T$, used in multigrid)
- Future FFT
- Out-of-core or I/O-intensive data analysis and algorithms
- Block iterative solvers (convergence & performance trade-offs)
- Sparse LU
- Data structures and algorithms (trees, graphs)
- Look at mixed-precision
- Discrete-event approaches to continuous systems simulation
- Automated performance analysis and modeling, tuning
- “Unconventional,” but related
    - Distributed deadlock detection for MPI
    - UPC language extensions (dynamic block sizes)
    - Exact linear algebra

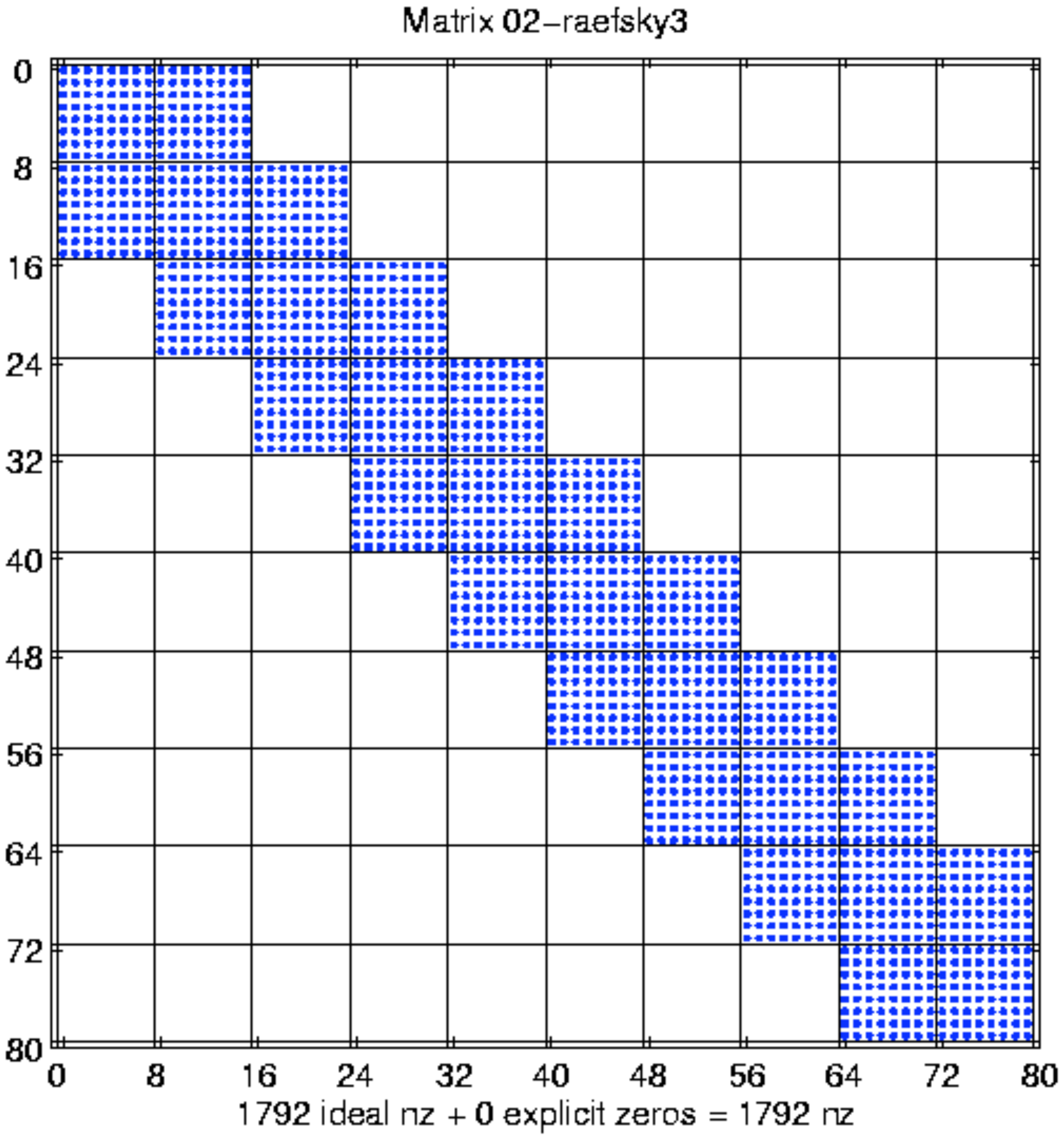
Sparse linear algebra

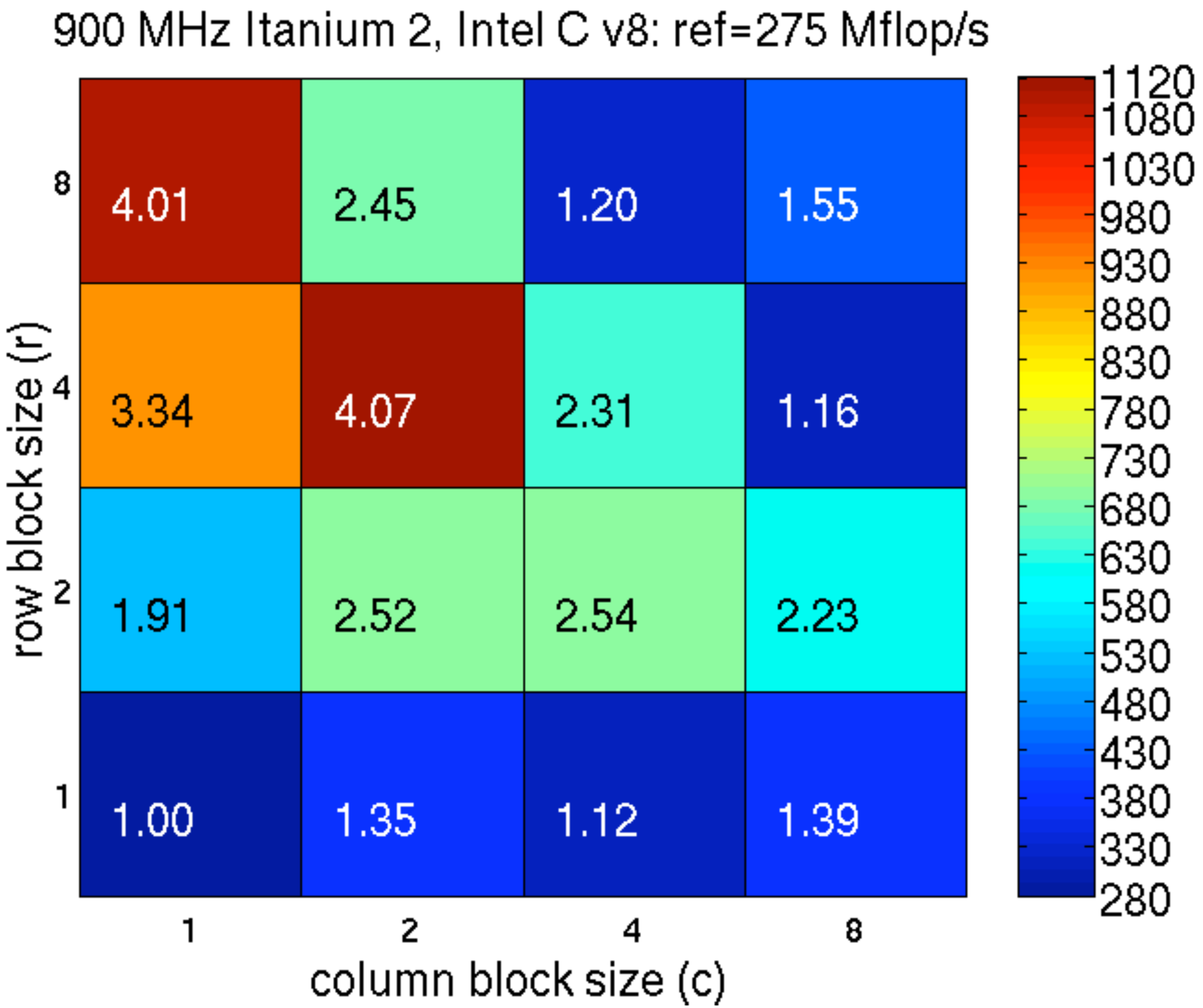
Key distinctions in autotuning work for sparse kernels

- Data structure transformations
  - Recall HW1
    - Sparse data structures require meta-data overhead
    - Sparse matrix-vector multiply (SpMV) is memory bound
  - Bandwidth limited ⇒ minimize data structure size
- Run-time tuning: Need lightweight techniques
- Extra flops pay off

Sparsity (1998) and OSKI (2005)

- Berkeley projects (BeBOP group: Demmel & Yelick; Im, Vuduc, *et al*.)
  - PHiPAC ⇒ SPARSITY ⇒ OSKI
  - On-going: See multicore optimizations by Williams, *et al*., in SC 2007
- Motivation: Sparse matrix-vector multiply (SpMV) ≤ 10% peak or less
  - Indirect, irregular memory access
  - Low q vs. dense case
  - Depends on machine and matrix, possibly unknown until run-time

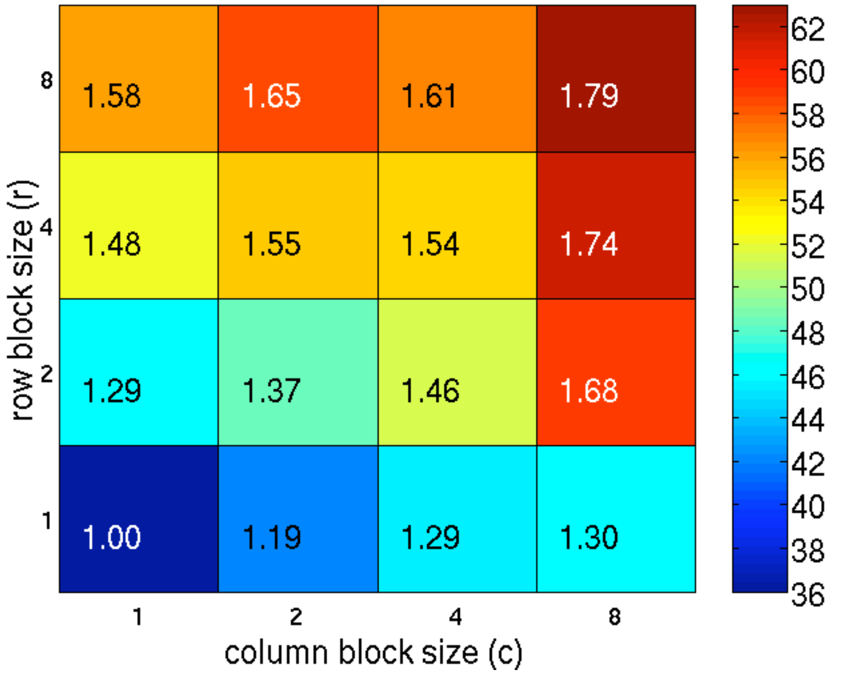

Matrix 02−raefsky3
0
8
16
24
32
40
48
56
64
72
80
0
8
16
24
32
40
48
56
64
72
80
1792 ideal nz + 0 explicit zeros = 1792 nz



900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s

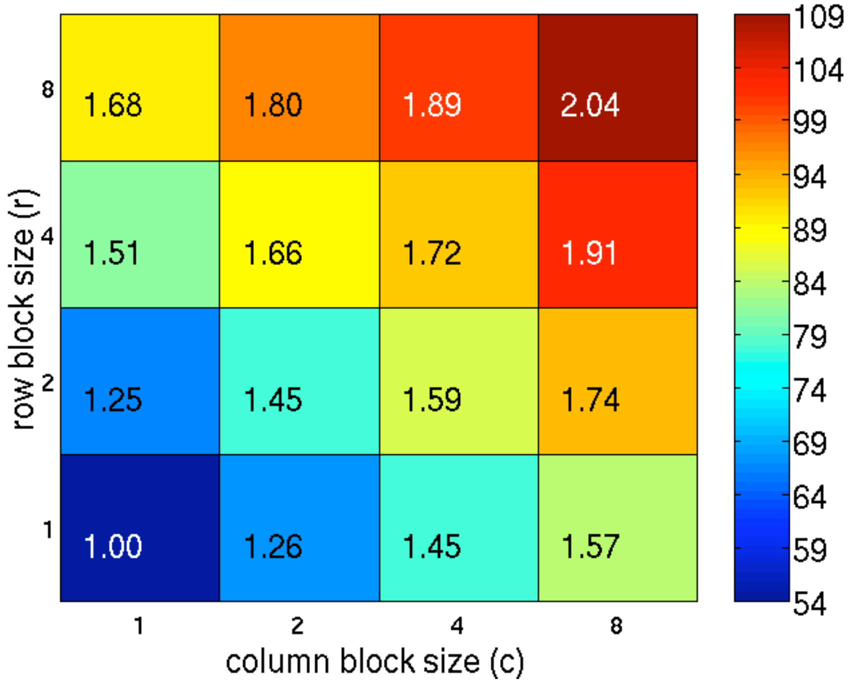
row block size (r) \ column block size (c)	1	2	4	8
8	4.01	2.45	1.20	1.55
4	3.34	4.07	2.31	1.16
2	1.91	2.52	2.54	2.23
1	1.00	1.35	1.12	1.39

333 MHz Sun Ultra 2i, Sun C v6.0: ref=35 Mflop/s



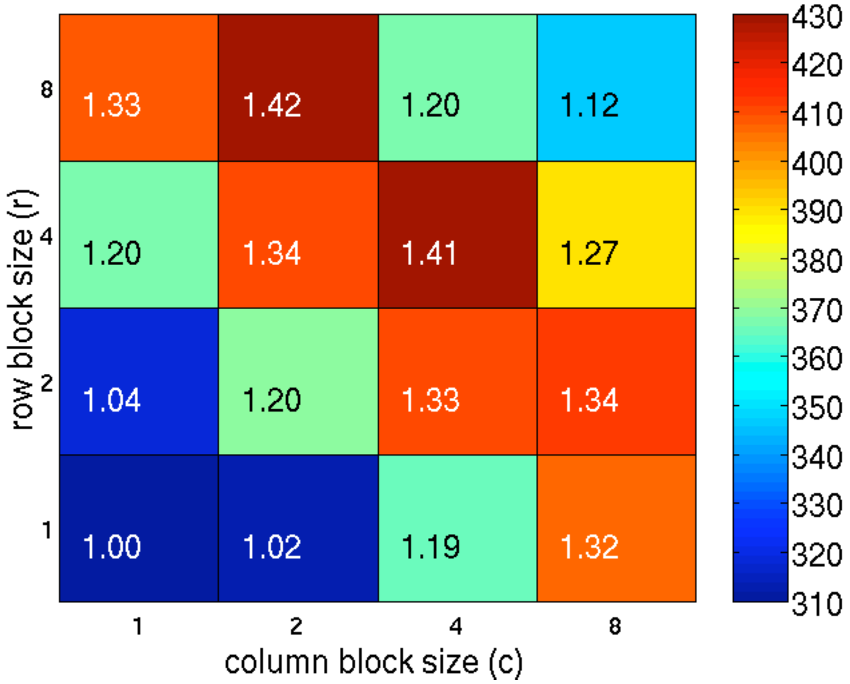
r \ c	1	2	4	8
8	1.58	1.65	1.61	1.79
4	1.48	1.55	1.54	1.74
2	1.29	1.37	1.46	1.68
1	1.00	1.19	1.29	1.30

900 MHz Ultra 3, Sun CC v6: ref=54 Mflop/s



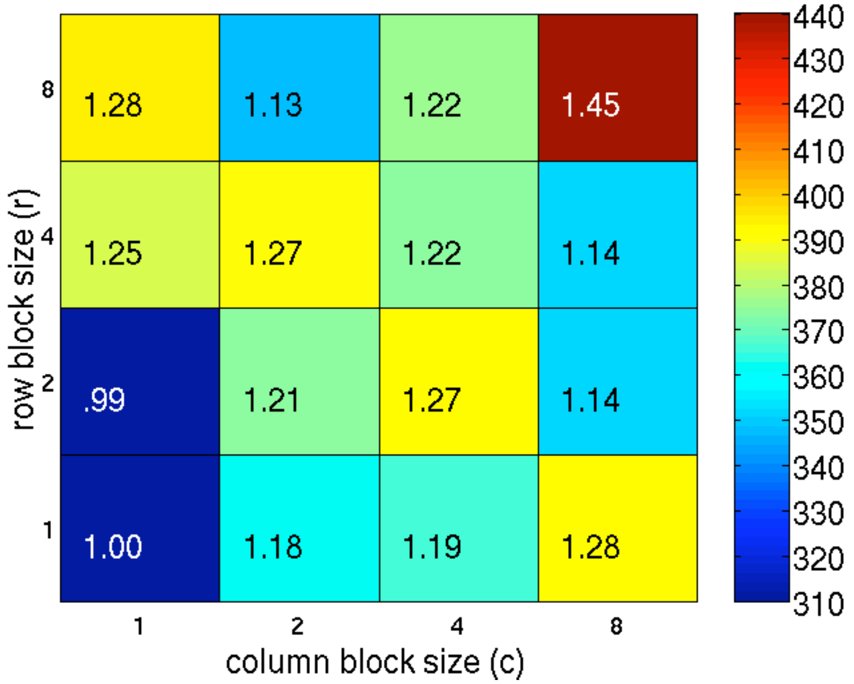
r \ c	1	2	4	8
8	1.68	1.80	1.89	2.04
4	1.51	1.66	1.72	1.91
2	1.25	1.45	1.59	1.74
1	1.00	1.26	1.45	1.57

2 GHz Pentium M, Intel C v8.1: ref=308 Mflop/s

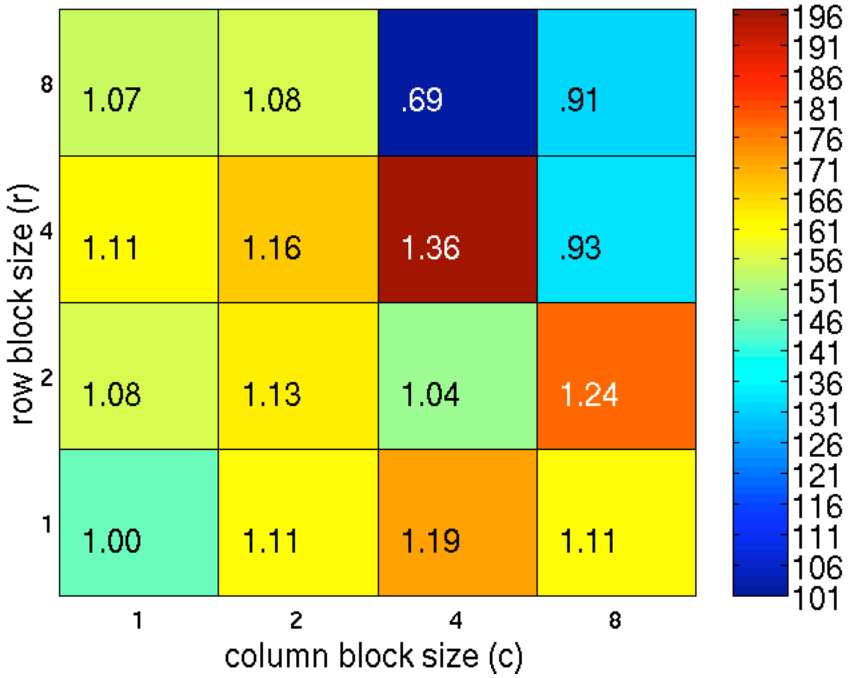


r \ c	1	2	4	8
8	1.33	1.42	1.20	1.12
4	1.20	1.34	1.41	1.27
2	1.04	1.20	1.33	1.34
1	1.00	1.02	1.19	1.32

1.4 GHz Opteron, gcc 3.4.2: ref=308 Mflop/s

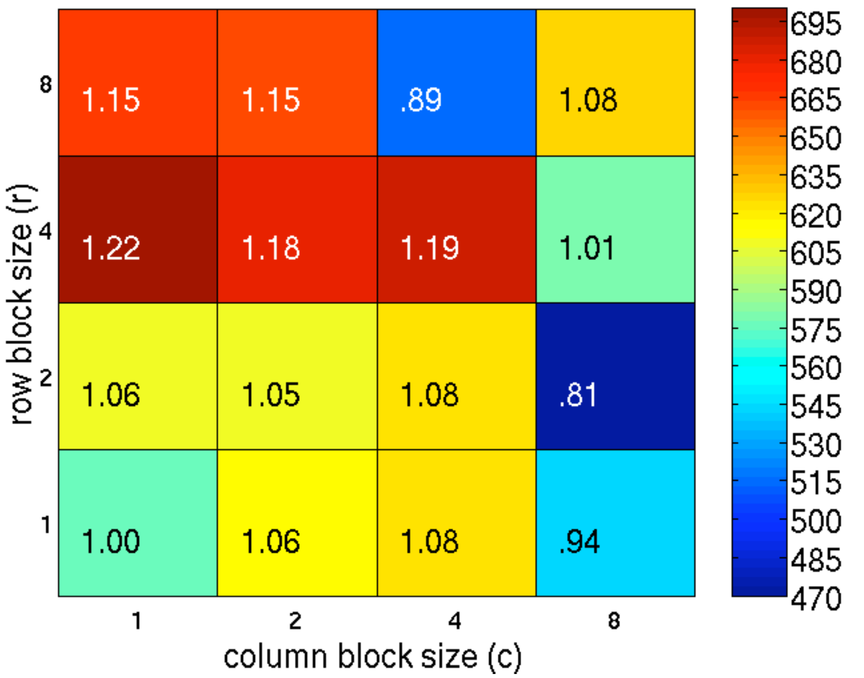


r \ c	1	2	4	8
8	1.28	1.13	1.22	1.45
4	1.25	1.27	1.22	1.14
2	.99	1.21	1.27	1.14
1	1.00	1.18	1.19	1.28



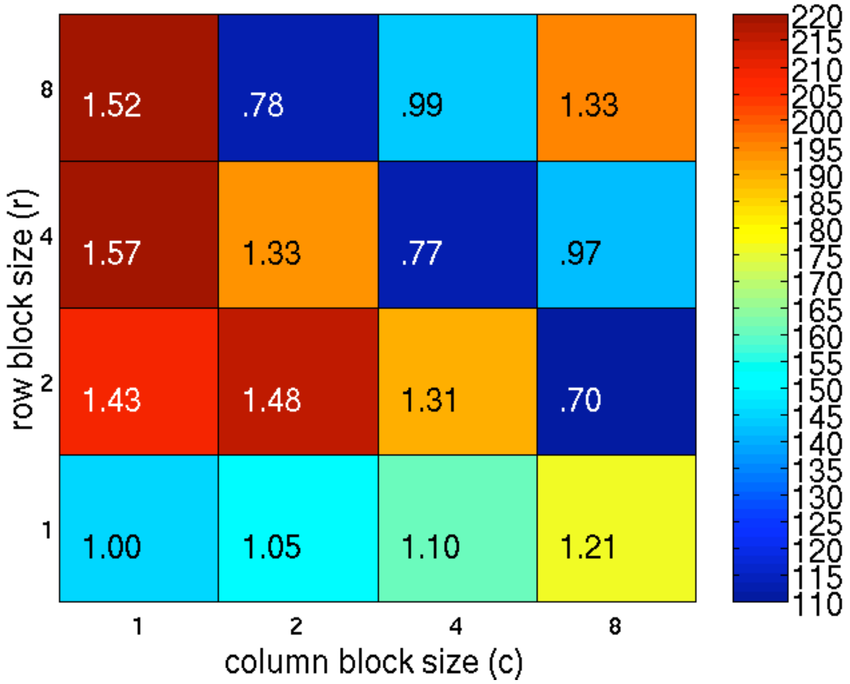
375 MHz Power3, IBM xlc v6: ref=145 Mflop/s

row block size (r) \ column block size (c)	1	2	4	8
8	1.07	1.08	.69	.91
4	1.11	1.16	1.36	.93
2	1.08	1.13	1.04	1.24
1	1.00	1.11	1.19	1.11



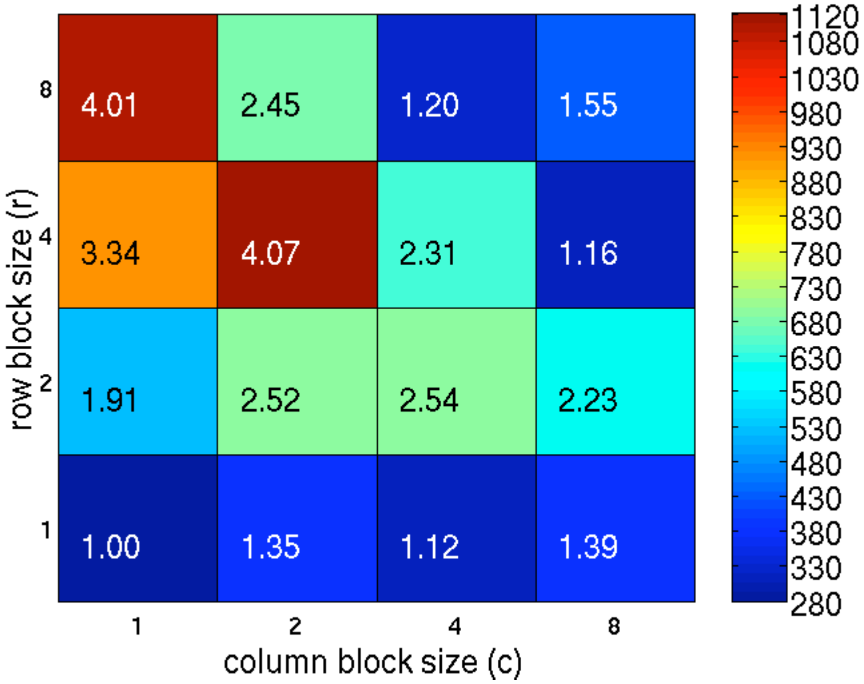
1.3 GHz Power4, IBM xlc v6: ref=577 Mflop/s

row block size (r) \ column block size (c)	1	2	4	8
8	1.15	1.15	.89	1.08
4	1.22	1.18	1.19	1.01
2	1.06	1.05	1.08	.81
1	1.00	1.06	1.08	.94



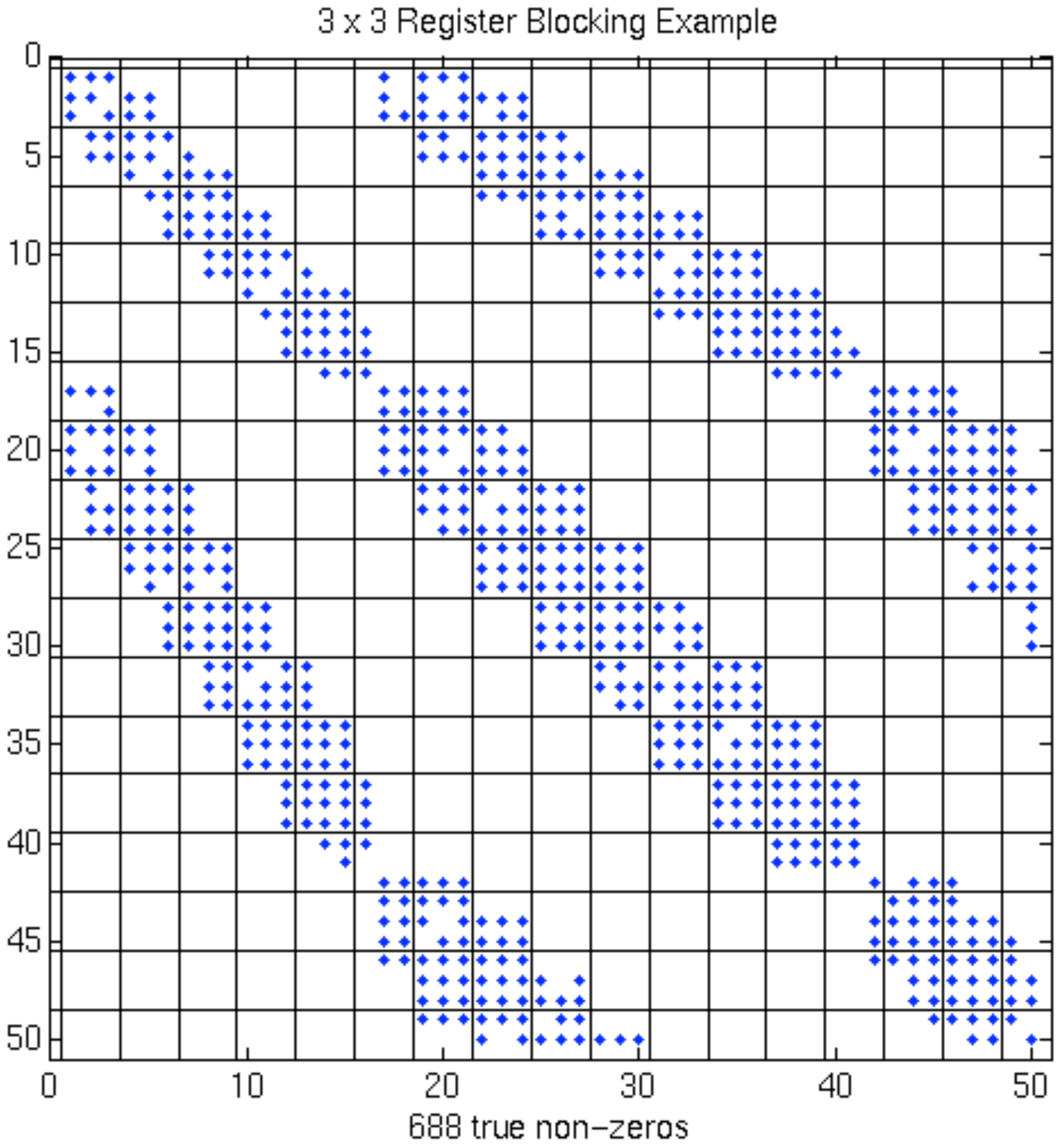
800 MHz Itanium, Intel C v7: ref=146 Mflop/s

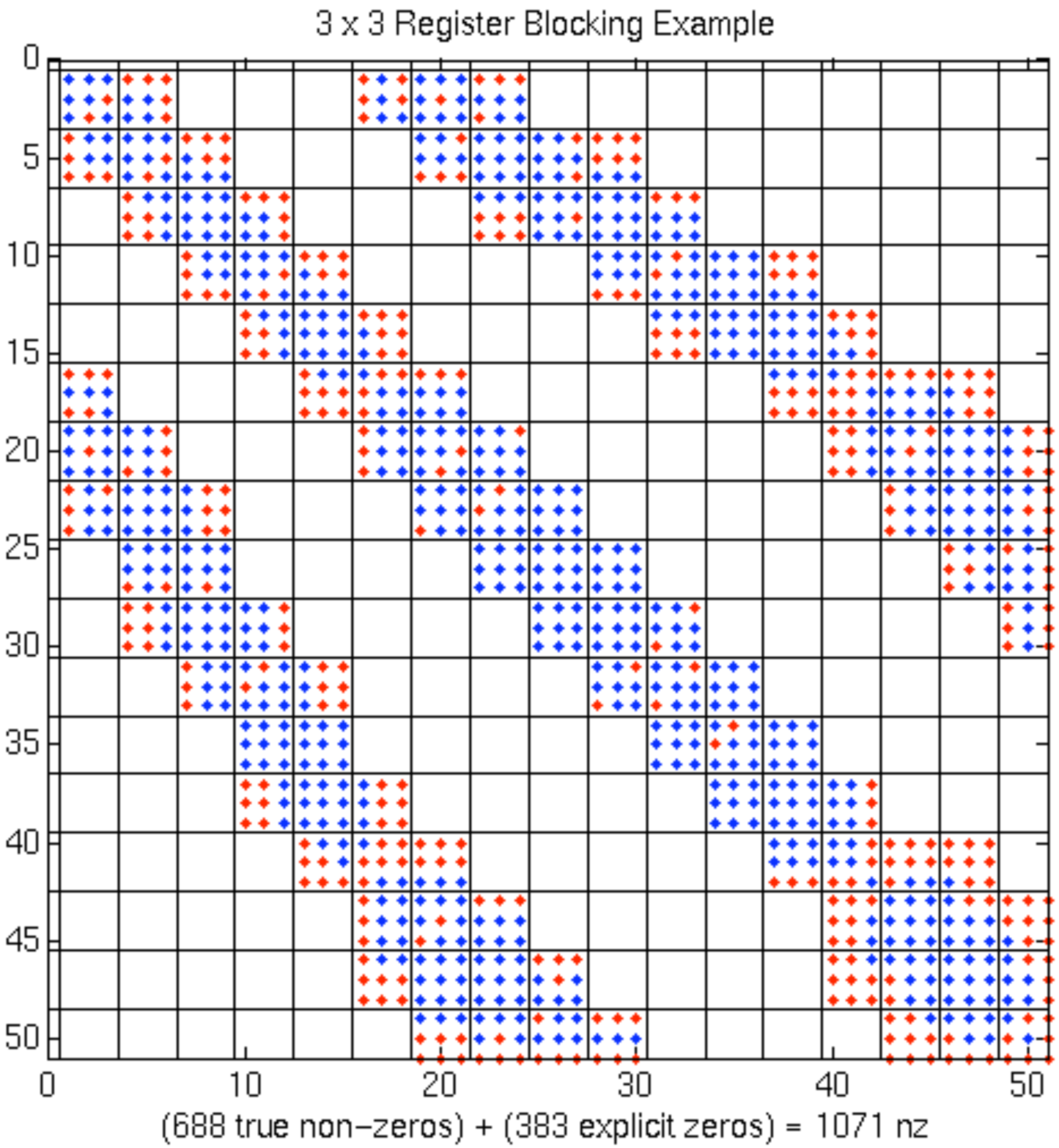
row block size (r) \ column block size (c)	1	2	4	8
8	1.52	.78	.99	1.33
4	1.57	1.33	.77	.97
2	1.43	1.48	1.31	.70
1	1.00	1.05	1.10	1.21



900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s

row block size (r) \ column block size (c)	1	2	4	8
8	4.01	2.45	1.20	1.55
4	3.34	4.07	2.31	1.16
2	1.91	2.52	2.54	2.23
1	1.00	1.35	1.12	1.39


3 x 3 Register Blocking Example
0
5
10
15
20
25
30
35
40
45
50
0
10
20
30
40
50
688 true non−zeros



50% extra zeros

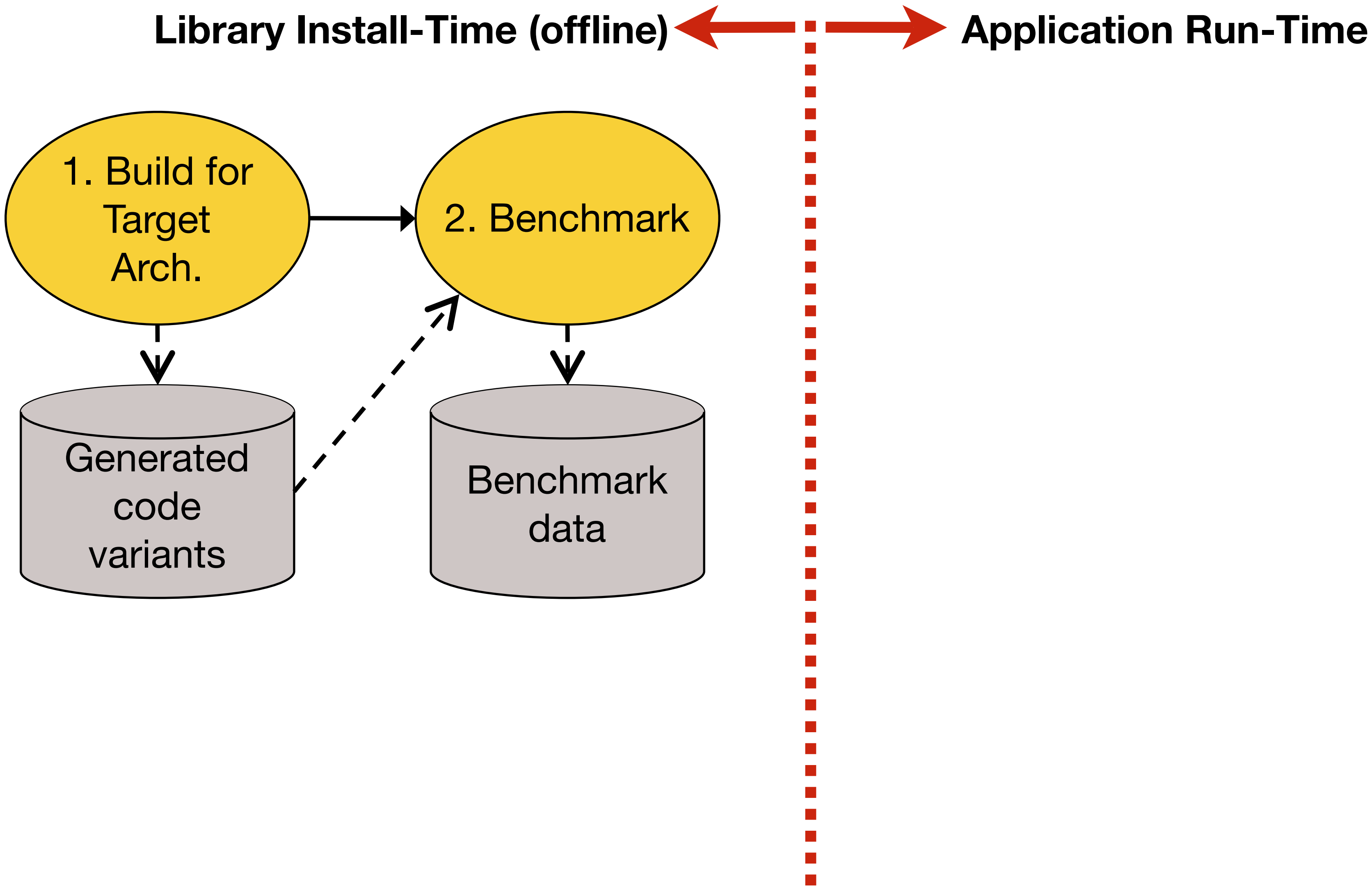
1.5x faster (2/3 time) on Pentium III

How OSKI tunes

Library Install-Time (offline)

Application Run-Time

How OSKI tunes



Library Install-Time (offline)

Application Run-Time

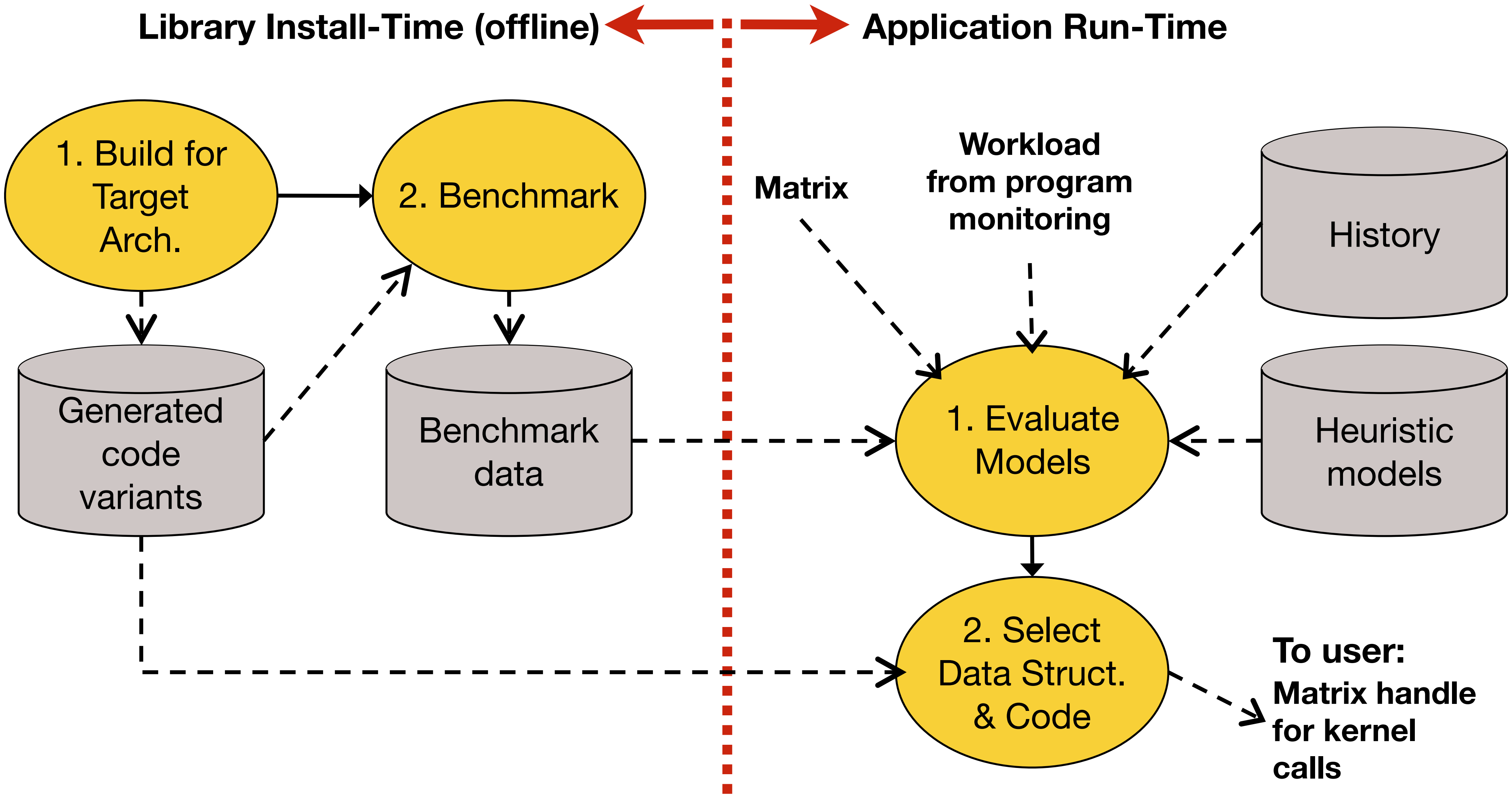
1. Build for Target Arch.

2. Benchmark

Generated code variants

Benchmark data

How OSKI tunes



Heuristic model example: Selecting a block size

- Idea: Hybrid off-line/run-time model
  - **Offline benchmark**: Measure Mflops(r, c) on dense matrix in sparse format
  - **Run-time**: Sample matrix to quickly estimate Fill(r, c)
  - Run-time **model**: Choose r, c to maximize Mflops(r,c) / Fill(r, c)
  - Accurate in practice (selects r x c with performance within 10% of best)
- Run-time **cost**?
  - Roughly 40 SpMVs
  - Dominated by conversion (~80%)

Workload tuning

- Consider BiCG solver: Equal mix of A^*x and $A^{T*}y$ (independent)
  - 3×1: A·x, AT·y = 1053, 343 Mflop/s ⇒ 517 Mflop/s
  - 3×3: A·x, AT·y = 806, 826 Mflop/s ⇒ 816 Mflop/s
- Higher-level operation: Fused (A^*x, $A^{T*}y$) kernel
  - 3×1: 757 Mflop/s
  - 3×3: 1400 Mflop/s

Tensor Contraction Engine (TCE) for quantum chemistry

Tensor Contraction Engine (TCE)

- Application domain: Quantum chemistry
  - Electronic structure calculations
  - Dominant computation expressible as a “tensor contraction”
- TCE generates a complete parallel program from a high-level spec
  - Automates time-space trade-offs
  - Output
- S. Hirata (2002), and many others
- Following presentation taken from Proc. IEEE 2005 special issue

Motivation: Simplify program development

```
hbar[a,b,i,j] == sum[f[b,c] * t[i,j,a,c], c] - sum[f[k,c] * t[k,b] * t[i,j,a,c], k,c] + sum[f[a,c] * t[i,j,c,b], c] - sum[f[k,c] * t[k,a] * t[i,j,c,b], k,c] - sum[f[k,j] * t[i,k,a,b], k] - sum[f[k,c] *
t[j,c] * t[i,k,a,b], k,c] - sum[f[k,i] * t[j,k,b,a], k] - sum[f[k,c] * t[i,c] * t[j,k,b,a], k,c] + sum[t[i,c] * t[j,d] * v[a,b,c,d], c,d] + sum[t[i,j,c,d] * v[a,b,c,d], c,d] + sum[t[j,c] * v[a,b,i,c],
c] - sum[t[k,b] * v[a,k,i,j], k] + sum[t[i,c] * v[b,a,j,c], c] - sum[t[k,a] * v[b,k,j,i], k] - sum[t[k,d] * t[i,j,c,b] * v[k,a,c,d], k,c,d] - sum[t[i,c] * t[j,k,b,d] * v[k,a,c,d], k,c,d] - sum[t[j,c]
* t[k,b] * v[k,a,c,i], k,c] + 2 * sum[t[j,k,b,c] * v[k,a,c,i], k,c] - sum[t[j,k,c,b] * v[k,a,c,i], k,c] - sum[t[i,c] * t[j,d] * t[k,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[k,d] * t[i,j,c,b] * v[k,a,d,c],
k,c,d] - sum[t[k,b] * t[i,j,c,d] * v[k,a,d,c], k,c,d] - sum[t[j,d] * t[i,k,c,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[i,c] * t[j,k,b,d] * v[k,a,d,c], k,c,d] - sum[t[i,c] * t[j,k,d,b] * v[k,a,d,c], k,c,d] -
sum[t[j,k,b,c] * v[k,a,i,c], k,c] - sum[t[i,c] * t[k,b] * v[k,a,j,c], k,c] - sum[t[i,k,c,b] * v[k,a,j,c], k,c] - sum[t[i,c] * t[j,d] * t[k,a] * v[k,b,c,d], k,c,d] - sum[t[k,d] * t[i,j,a,c] * v[k,b,c,d],
k,c,d] - sum[t[k,a] * t[i,j,c,d] * v[k,b,c,d], k,c,d] + 2 * sum[t[j,d] * t[i,k,a,c] * v[k,b,c,d], k,c,d] - sum[t[j,d] * t[i,k,c,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[j,k,d,a] * v[k,b,c,d], k,c,d]
- sum[t[i,c] * t[k,a] * v[k,b,c,j], k,c] + 2 * sum[t[i,k,a,c] * v[k,b,c,j], k,c] - sum[t[i,k,c,a] * v[k,b,c,j], k,c] + 2 * sum[t[k,d] * t[i,j,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,d] * t[i,k,a,c] *
v[k,b,d,c], k,c,d] - sum[t[j,c] * t[k,a] * v[k,b,i,c], k,c] - sum[t[j,k,c,a] * v[k,b,i,c], k,c] - sum[t[i,k,a,c] * v[k,b,j,c], k,c] + sum[t[i,c] * t[j,d] * t[k,a] * t[l,b] * v[k,l,c,d], k,l,c,d] - 2 *
sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[k,l,c,d], k,l,c,d] + sum[t[k,a] * t[l,b] * t[i,j,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,c] * t[l,d] *
t[i,k,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,c,d], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,c,a] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[k,l,c,d],
k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,b] * t[j,k,d,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,c,d], k,l,c,d] + 4 * sum[t[i,k,a,c] *
t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d]
+ sum[t[i,k,c,a] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[j,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,j,c,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,c,b] * t[k,l,a,d] *
v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,a,c] * t[k,l,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[j,c] * t[k,b] * t[l,a] * v[k,l,c,i], k,l,c] + sum[t[l,c] * t[j,k,b,a] * v[k,l,c,i], k,l,c] - 2 * sum[t[l,a] * t[j,k,b,c]
* v[k,l,c,i], k,l,c] + sum[t[l,a] * t[j,k,c,b] * v[k,l,c,i], k,l,c] - 2 * sum[t[k,c] * t[j,l,b,a] * v[k,l,c,i], k,l,c] + sum[t[k,a] * t[j,l,b,c] * v[k,l,c,i], k,l,c] + sum[t[k,b] * t[j,l,c,a] * v[k,l,c,i],
k,l,c] + sum[t[j,c] * t[l,k,a,b] * v[k,l,c,i], k,l,c] + sum[t[i,c] * t[k,a] * t[l,b] * v[k,l,c,j], k,l,c] + sum[t[l,c] * t[i,k,a,b] * v[k,l,c,j], k,l,c] - 2 * sum[t[l,b] * t[i,k,a,c] * v[k,l,c,j], k,l,c]
+ sum[t[l,b] * t[i,k,c,a] * v[k,l,c,j], k,l,c] + sum[t[i,c] * t[k,l,a,b] * v[k,l,c,j], k,l,c] + sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,d,c],
k,l,c,d] + sum[t[j,d] * t[l,a] * t[i,k,c,b] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,a] *
t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,b] * t[j,l,d,a] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,d,c], k,l,c,d] + sum[t[k,a]
* t[l,b] * v[k,l,i,j], k,l] + sum[t[k,l,a,b] * v[k,l,i,j], k,l] + sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[l,k,c,d], k,l,c,d] + sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,d] *
t[j,k,b,a] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,d,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,c,b] * t[k,l,a,d] * v[l,k,c,d], k,l,c,d]
+ sum[t[i,j,a,c] * t[k,l,b,d] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[l,c] * t[i,k,a,b] * v[l,k,c,j], k,l,c] + sum[t[l,b] * t[i,k,a,c] * v[l,k,c,j], k,l,c] + sum[t[l,a] * t[i,k,c,b] * v[l,k,c,j], k,l,c] +
v[a,b,i,j]
```

Source: Baumgartner, et al. *(2005)*

Rewriting to reduce operation counts

Naïvely, ≈ 4 × N^{10} flops

$$S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}$$

$$\Downarrow$$

$$S_{abij} = \sum_{c,k} \left(\sum_{d,f} \left(\sum_{e,l} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

Assuming associativity and distributivity, ≈ 6 × N^6 flops, but also requires temporary storage.

Source: Baumgartner, et al. *(2005)*

Operation and storage minimization *via* loop fusion

$$\begin{aligned} T^{(1)}_{bcdf} &= \sum_{e,l} B_{befl} \times D_{cdel} \\ T^{(2)}_{bcjk} &= \sum_{d,f} T^{(1)}_{bcdf} \times C_{dfjk} \\ S_{abij} &= \sum_{c,k} T^{(2)}_{bcjk} \times A_{acik} \end{aligned}$$

```
$T1 = T2 = S = 0$
for $b, c, d, e, f, l$ do
    $T1[b, c, d, f] \mathrel{+}= B[b, e, f, l] \cdot D[c, d, e, l]$
for $b, c, d, f, j, k$ do
    $T2[b, c, j, k] \mathrel{+}= T1[b, c, d, f] \cdot C[d, f, j, k]$
for $a, b, c, i, j, k$ do
    $S[a, b, i, j] \mathrel{+}= T2[b, c, j, k] \cdot A[a, c, i, k]$
```

Operation and storage minimization *via* loop fusion

$$T^{(1)}_{bcdf} = \sum_{e,l} B_{befl} \times D_{cdel}$$

$$T^{(2)}_{bcjk} = \sum_{d,f} T^{(1)}_{bcdf} \times C_{dfjk}$$

$$S_{abij} = \sum_{c,k} T^{(2)}_{bcjk} \times A_{acik}$$

```
$T1 = T2 = S = 0$
for $b, c, d, e, f, l$ do
  $T1[b, c, d, f] \mathrel{+}= B[b, e, f, l] \cdot D[c, d, e, l]$
for $b, c, d, f, j, k$ do
  $T2[b, c, j, k] \mathrel{+}= T1[b, c, d, f] \cdot C[d, f, j, k]$
for $a, b, c, i, j, k$ do
  $S[a, b, i, j] \mathrel{+}= T2[b, c, j, k] \cdot A[a, c, i, k]$
```

```
$S = 0$
for $b, c$ do
  $T1f \leftarrow 0,\ T2f \leftarrow 0$
  for $d, f$ do
    for $e, l$ do
      $T1f \mathrel{+}= B[b, e, f, l] \cdot D[c, d, e, l]$
    for $j, k$ do
      $T2f[j, k] \mathrel{+}= T1f \cdot C[d, f, j, k]$
  for $a, i, j, k$ do
    $S[a, b, i, j] \mathrel{+}= T2f[j, k] \cdot A[a, c, i, k]$
```

Time-space trade-offs

Max index of
***a—f*: *O*(1000)**
***i—k*: *O*(100)**

```
for $a, e, c, f$ do
    for $i, j$ do
        $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$   ← “Contraction” of T over i, j
for $c, e, b, k$ do
    $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, f, b, k$ do
    $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
                                   Integrals, O(1000) flops
for $c, e, a, f$ do
    for $b, k$ do
        $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$   ← “Contraction” over $T^{(1)}$ and $T^{(2)}$
for $c, e, a, f$ do
    $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

Time-space trade-offs

Max index of
a—*f*: *O*(1000)
i—*k*: *O*(100)

```
for $a, e, c, f$ do ←
    for $i, j$ do
        $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
for $c, e, b, k$ do
    $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, f, b, k$ do
    $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
for $c, e, a, f$ do ←
    for $b, k$ do
        $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
for $c, e, a, f$ do ←
    $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

Same indices
⇒ Loop fusion candidates

Time-space trade-offs

```
for $a, e, c, f$ do
  for $i, j$ do
    $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
for $c, e, b, k$ do
  $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, f, b, k$ do
  $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
for $c, e, a, f$ do
  for $b, k$ do
    $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
for $c, e, a, f$ do
  $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

```
for $a, e, c, f$ do
  for $i, j$ do
    $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
for $a, c, e, f, b, k$ do
  $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, e, c, f, b, k$ do
  $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
for $c, e, a, f$ do
  for $b, k$ do
    $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
for $c, e, a, f$ do
  $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

Add extra flops

Time-space trade-offs

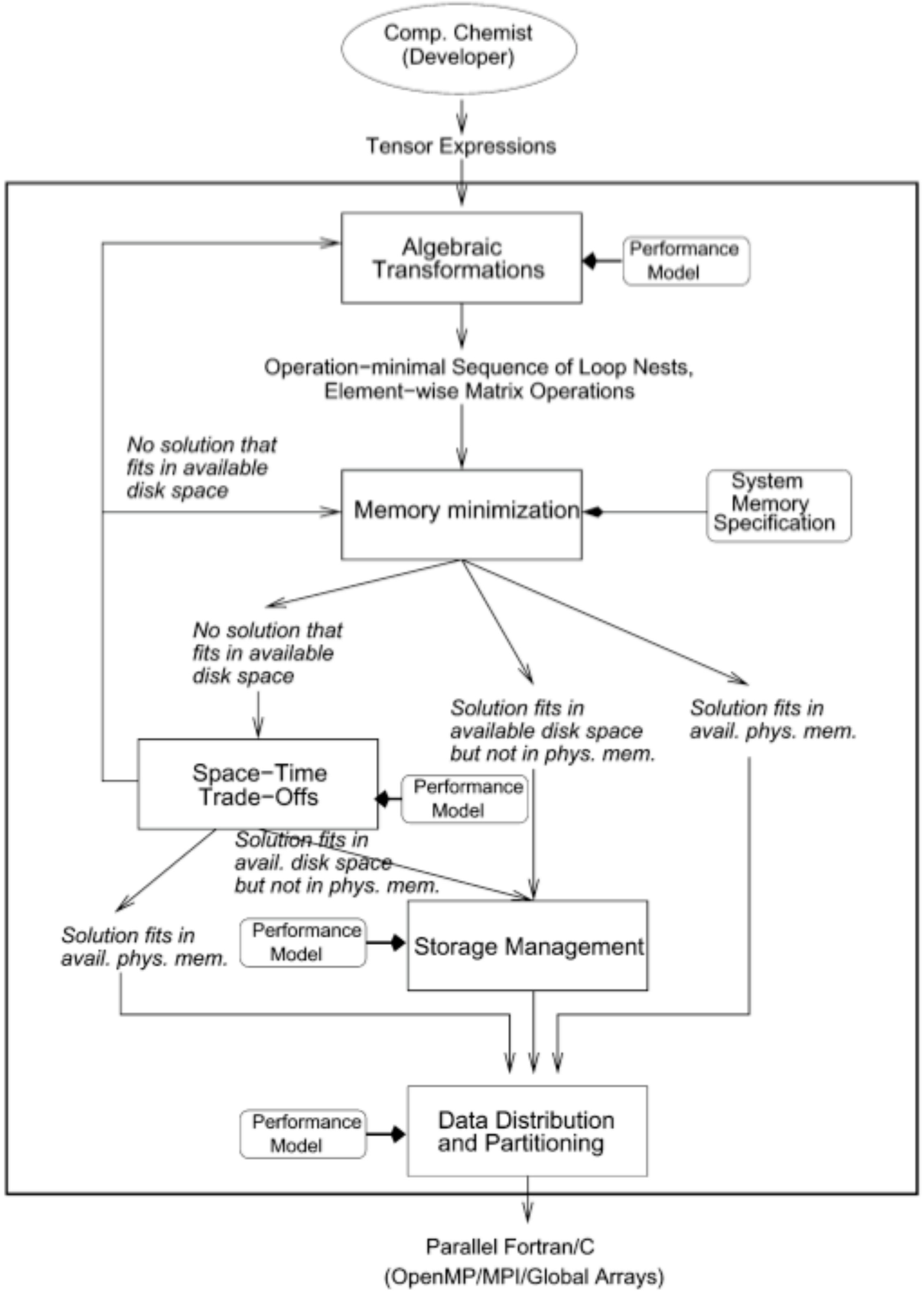
```
for $a, e, c, f$ do ⟶
    for $i, j$ do
        $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
for $c, e, b, k$ do
    $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, f, b, k$ do
    $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
for $c, e, a, f$ do ⟶
    for $b, k$ do
        $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
for $c, e, a, f$ do ⟶
    $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

```
for $a, e, c, f$ do        ⇐ Fused
    for $i, j$ do
        $x \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
    for $b, k$ do
        $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
        $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
        $y \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
    $E \mathrel{+}= x \cdot y$
```

```
for $a, e, c, f$ do
    for $i, j$ do
        $X_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
for $c, e, b, k$ do
    $T^{(1)}_{cebk} \leftarrow f_1(c, e, b, k)$
for $a, f, b, k$ do
    $T^{(2)}_{afbk} \leftarrow f_2(a, f, b, k)$
for $c, e, a, f$ do
    for $b, k$ do
        $Y_{ceaf} \mathrel{+}= T^{(1)}_{cebk} \cdot T^{(2)}_{afbk}$
for $c, e, a, f$ do
    $E \mathrel{+}= X_{aecf} \cdot Y_{ceaf}$
```

Tiled & partially fused

```
for $a^B, e^B, c^B, f^B$ do
    for $a, e, c, f$ do
        for $i, j$ do
            $\hat{X}_{aecf} \mathrel{+}= T_{ijae} \cdot T_{ijcf}$
    for $b, k$ do
        for $c, e$ do
            $\hat{T}^{(1)}_{ce} \leftarrow f_1(c, e, b, k)$
        for $a, f$ do
            $\hat{T}^{(2)}_{af} \leftarrow f_2(a, f, b, k)$
        for $c, e, a, f$ do
            $\hat{Y}_{ceaf} \mathrel{+}= \hat{T}^{(1)}_{ce} \cdot \hat{T}^{(2)}_{af}$
    for $c, e, a, f$ do
        $E \mathrel{+}= \hat{X}_{aecf} \cdot \hat{Y}_{ceaf}$
```



Comp. Chemist (Developer)

Tensor Expressions

Algebraic Transformations

Performance Model

Operation−minimal Sequence of Loop Nests, Element−wise Matrix Operations

No solution that fits in available disk space

Memory minimization

System Memory Specification

No solution that fits in available disk space

Solution fits in available disk space but not in phys. mem.

Solution fits in avail. phys. mem.

Space−Time Trade−Offs

Performance Model

Solution fits in avail. disk space but not in phys. mem.

Solution fits in avail. phys. mem.

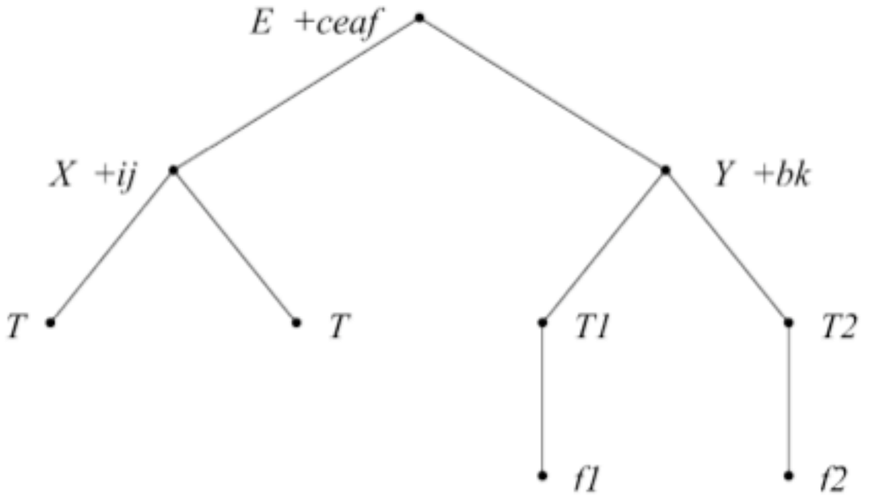
Performance Model

Storage Management

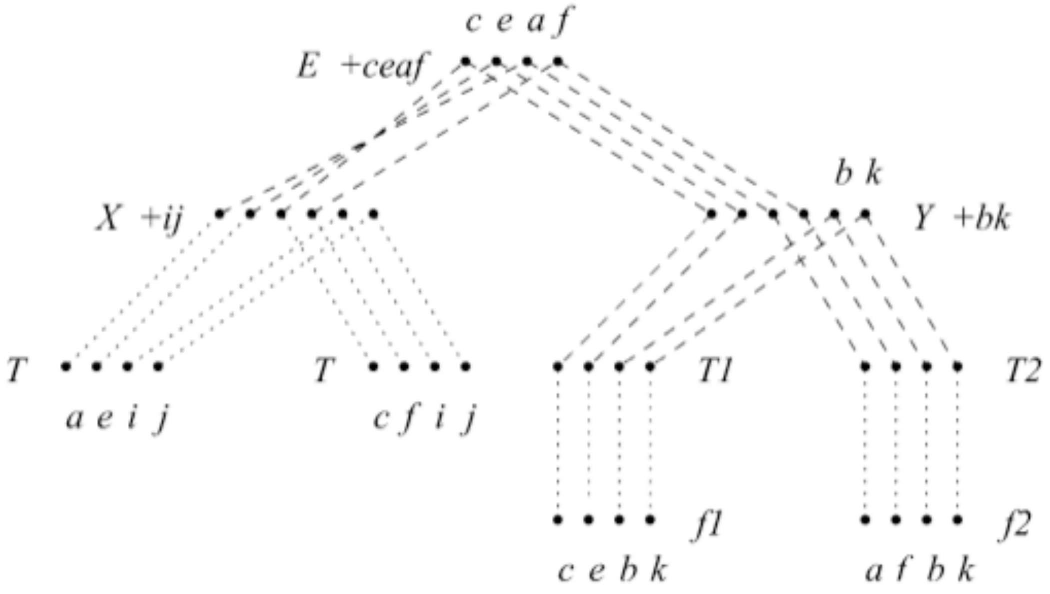
Performance Model

Data Distribution and Partitioning

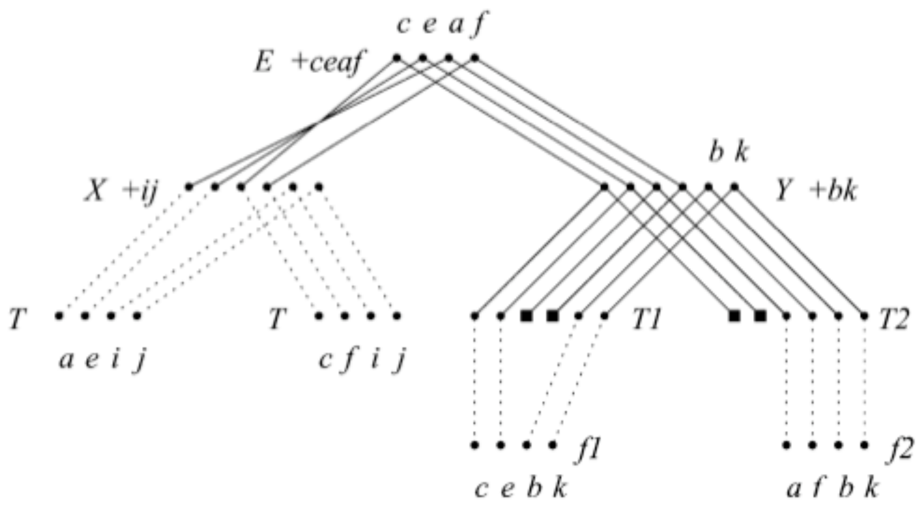
Parallel Fortran/C (OpenMP/MPI/Global Arrays)



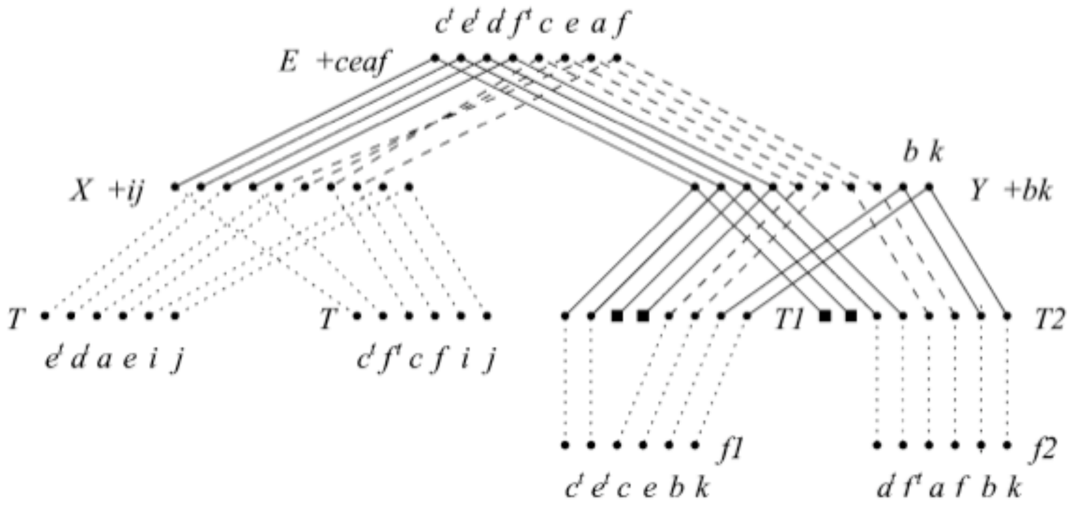
(a) Expression tree.



(b) Fusion graph.



(a) Fully fused computation from Fig. 4.



(b) Partially fused computation from Fig. 5.

Next time:
Empirical compilers and tools

“In conclusion…”

Backup slides