OSKI: A Library of Automatically Tuned Sparse Matrix Kernels

Richard Vuduc (LLNL), James Demmel, Katherine Yelick

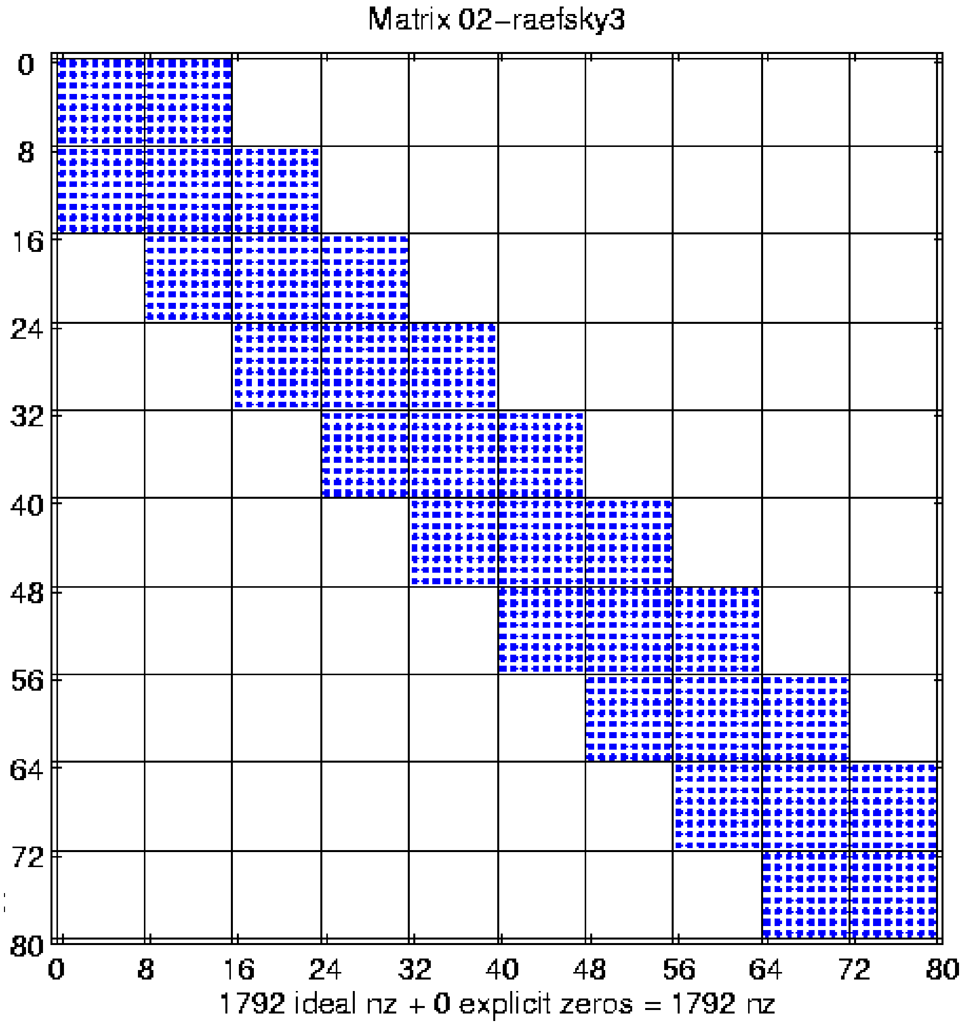
Berkeley Benchmarking and OPtimization (BeBOP) Project
bebop.cs.berkeley.edu
EECS Department, University of California, Berkeley

SIAM CSE
February 12, 2005

OSKI: Optimized Sparse Kernel Interface

- Sparse kernels tuned for user’s matrix & machine
  - Hides complexity of run-time tuning
  - Low-level BLAS-style functionality
    - Sparse matrix-vector multiply (SpMV), triangular solve (TrSV), …
  - Includes fast locality-aware kernels: A^TA*x, …
  - Initial target: cache-based superscalar uniprocessors
- Faster than standard implementations
  - Up to **4x** faster SpMV, **1.8x** TrSV, **4x** A^TA*x
- For “advanced” users & solver library writers
  - Available as stand-alone open-source library (pre-release)
  - PETSc extension in progress
- Written in C (can call from Fortran)

Motivation: The Difficulty of Tuning

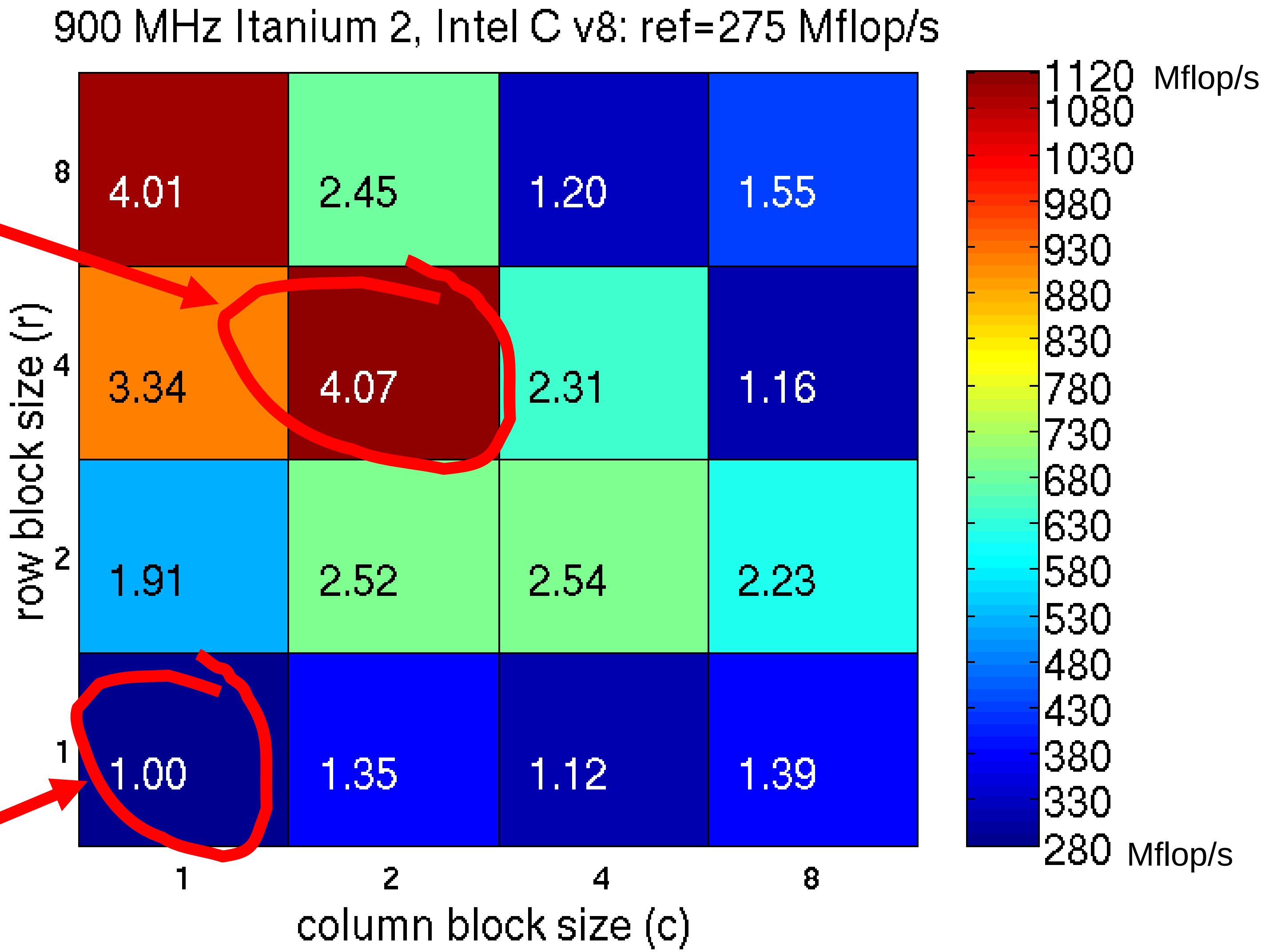


- n = 21216
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

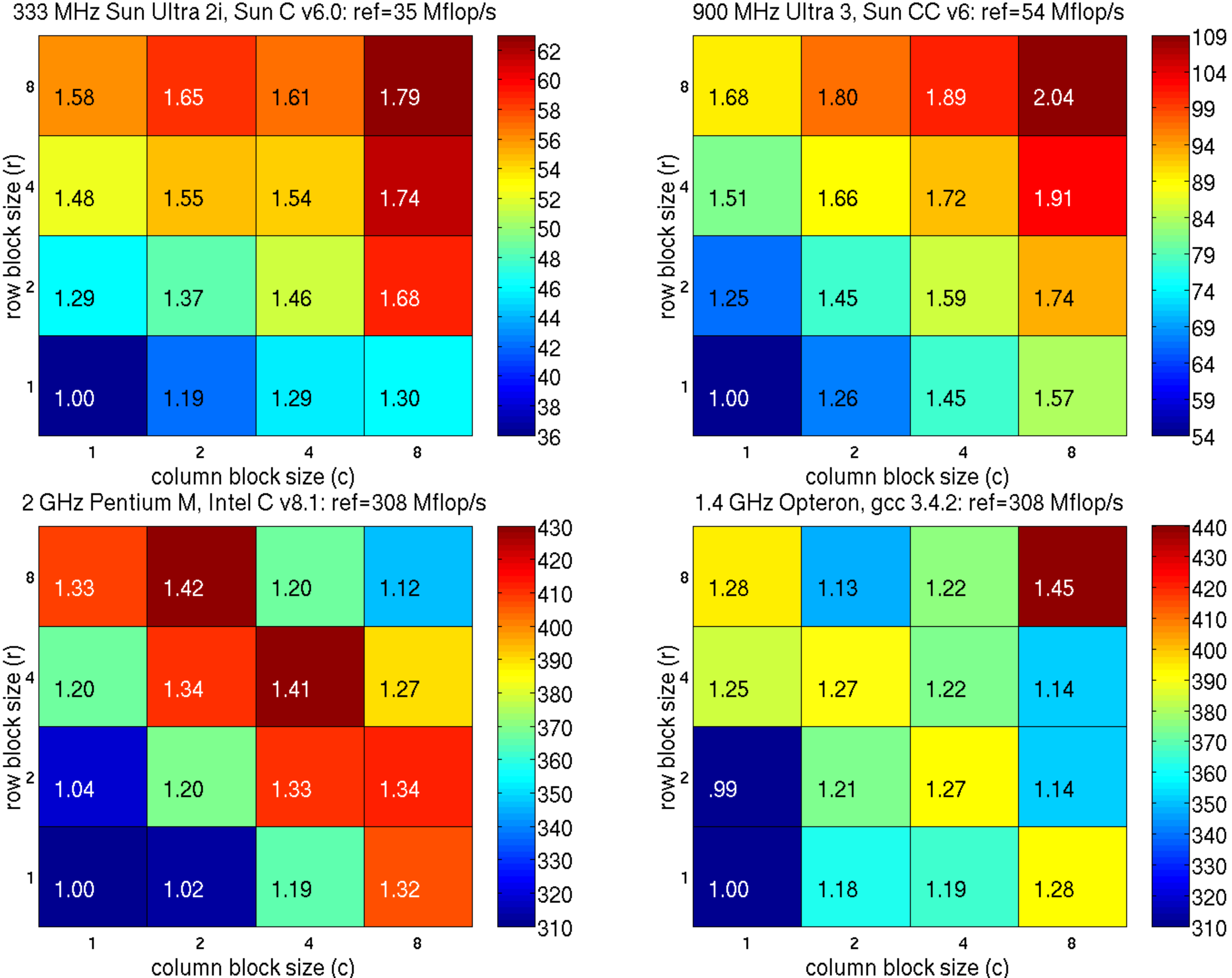
- **8x8** dense substructure

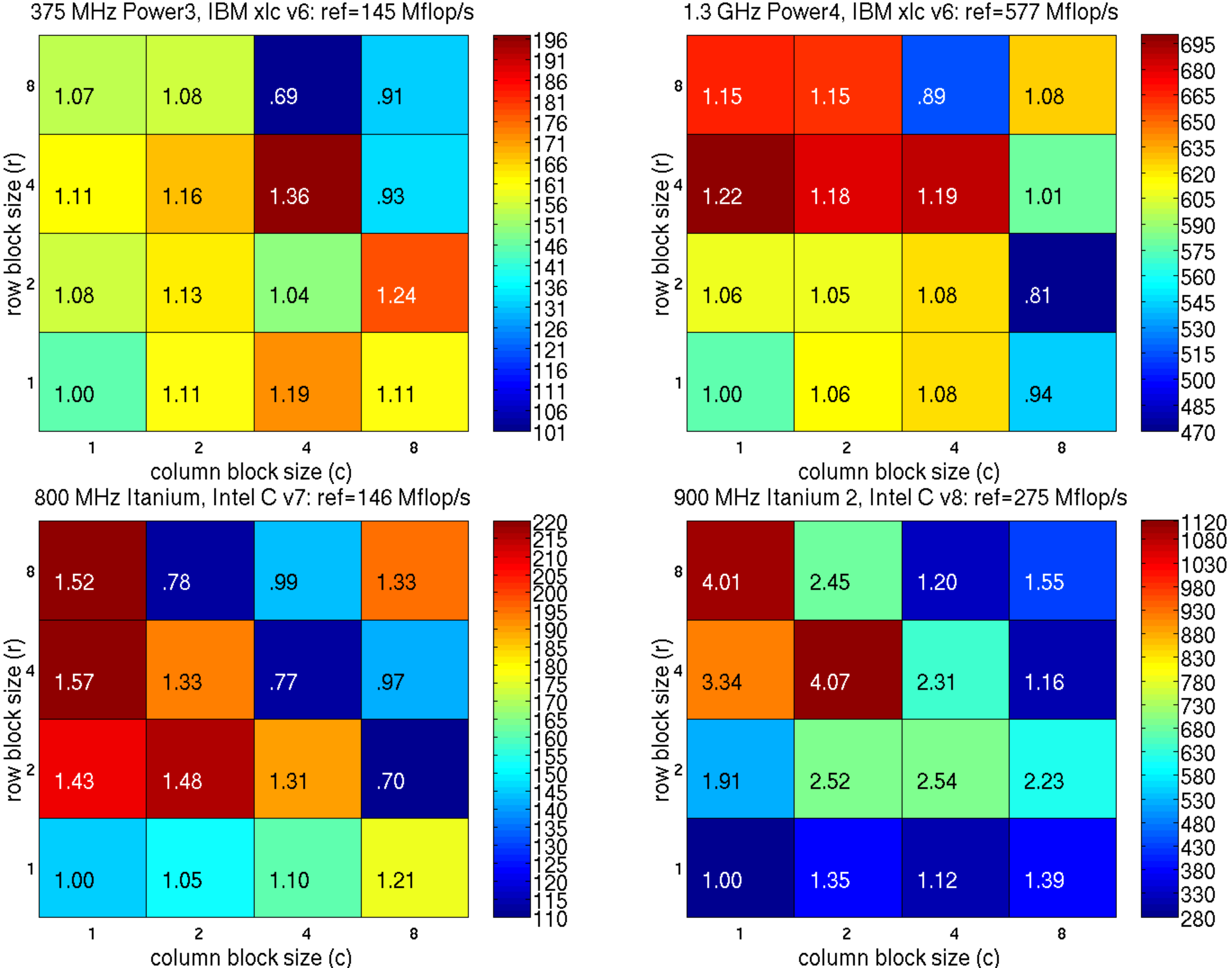
Speedups on Itanium 2: The Need for Search



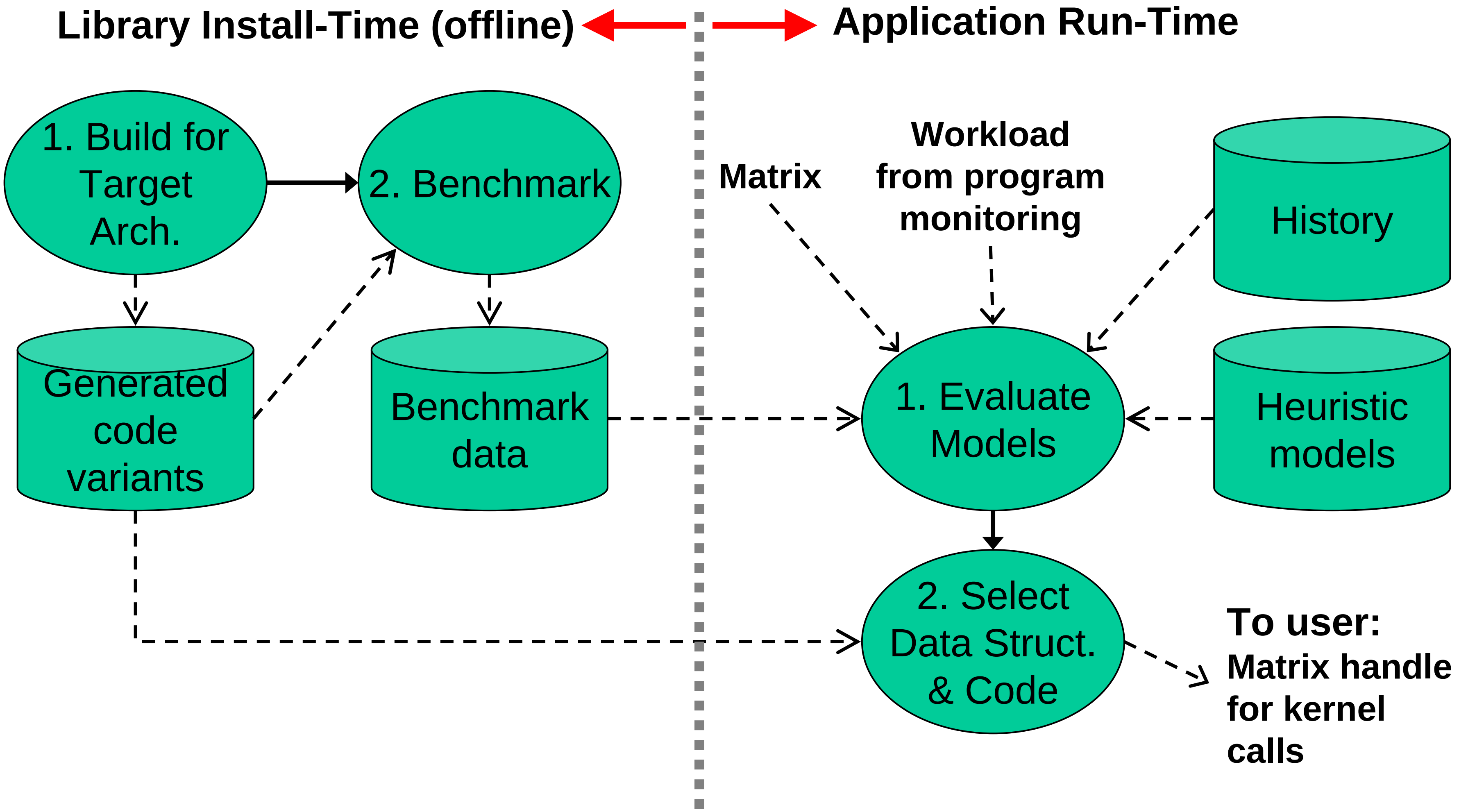
Best: 4x2

Reference


333 MHz Sun Ultra 2i, Sun C v6.0: ref=35 Mflop/s
row block size (r)
column block size (c)
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
1
2
4
8
62
60
58
56
54
52
50
48
46
44
42
40
38
36
900 MHz Ultra 3, Sun CC v6: ref=54 Mflop/s
row block size (r)
column block size (c)
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
1
2
4
8
109
104
99
94
89
84
79
74
69
64
59
54
2 GHz Pentium M, Intel C v8.1: ref=308 Mflop/s
row block size (r)
column block size (c)
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
2
4
8
430
420
410
400
390
380
370
360
350
340
330
320
310
1.4 GHz Opteron, gcc 3.4.2: ref=308 Mflop/s
row block size (r)
column block size (c)
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
1
2
4
8
440
430
420
410
400
390
380
370
360
350
340
330
320
310


375 MHz Power3, IBM xlc v6: ref=145 Mflop/s
row block size (r)
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
2
4
8
column block size (c)
196
191
186
181
176
171
166
161
156
151
146
141
136
131
126
121
116
111
106
101
1.3 GHz Power4, IBM xlc v6: ref=577 Mflop/s
row block size (r)
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
1
2
4
8
column block size (c)
695
680
665
650
635
620
605
590
575
560
545
530
515
500
485
470
800 MHz Itanium, Intel C v7: ref=146 Mflop/s
row block size (r)
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
1
2
4
8
column block size (c)
220
215
210
205
200
195
190
185
180
175
170
165
160
155
150
145
140
135
130
125
120
115
110
900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s
row block size (r)
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
1
2
4
8
column block size (c)
1120
1080
1030
980
930
880
830
780
730
680
630
580
530
480
430
380
330
280

How OSKI Tunes (Overview)



Library Install-Time (offline)

Application Run-Time

1. Build for Target Arch.

2. Benchmark

Generated code variants

Benchmark data

Matrix

Workload from program monitoring

History

1. Evaluate Models

Heuristic models

2. Select Data Struct. & Code

To user: Matrix handle for kernel calls

Extensibility: Advanced users may write & dynamically add “Code variants” and “Heuristic models” to system.

Cost of Tuning

- Non-trivial run-time tuning cost: up to ~40 mat-vecs
  - Dominated by conversion time
- Design point: user calls “tune” routine explicitly
  - Exposes cost
  - Tuning time limited using estimated workload
    - Provided by user or inferred by library
- User may save tuning results
  - To apply on future runs with similar matrix
  - Stored in “human-readable” format

How to Call OSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
```

```
/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
    my_matmult( ptr, ind, val, α, x, β, y );
```

How to Call OSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   my_matmult( ptr, ind, val, α, x, β, y );
```

How to Call OSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   oski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);/* Step 2 */
```

How to Call OSKI: Tune with Explicit Hints

- User calls “tune” routine
  - May provide explicit tuning hints (OPTIONAL)

```
oski_matrix_t A_tunable = oski_CreateMatCSR( … );
   /* … */
/* Tell OSKI we will call SpMV 500 times (workload hint) */
oski_SetHintMatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view, 500);
/* Tell OSKI we think the matrix has 8x8 blocks (structural hint) */
oski_SetHint(A_tunable, HINT_SINGLE_BLOCKSIZE, 8, 8);

oski_TuneMat(A_tunable); /* Ask OSKI to tune */

for( i = 0; i < 500; i++ )
   oski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```

How the User Calls OSKI: Implicit Tuning

- Ask library to infer workload
  - Library profiles all kernel calls
  - May periodically re-tune

```
oski_matrix_t A_tunable = oski_CreateMatCSR( … );
   /* … */

for( i = 0; i < 500; i++ ) {
   oski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
   oski_TuneMat(A_tunable); /* Ask OSKI to tune */
}
```

Additional Features

- Embedded scripting language for selecting customized, complex transformations
  - Mechanism to save/restore transformations

```
/*  In "my_app.c"  */
fp = fopen("my_xform.txt", "rt");
fgets(buffer, BUFSIZE, fp);

oski_ApplyMatTransform(A_tunable,
   buffer);
oski_MatMult(A_tunable, …);
```

```
#  In file, "my_xform.txt"
#  Compute A_fast = P*A*P^T using
     Pinar's reordering algorithm
A_fast, P =
   reorder_TSP(InputMat);
#  Split A_fast = A_1 + A_2, where A_1 in 2x2
     block format, A_2 in CSR
A1, A2 =
   A_fast.extract_blocks(2, 2);

return transpose(P)*(A1+A2)*P;
```

Additional Features

- GNU AutoTools (autoconf) based install process
- Support for several scalar type combinations
  - {32-bit, 64-bit int} x {single, double, complex, double_complex}
- “Plug-in” extensibility
  - Very advanced users may customize library (at run-time)
    - New heuristics
    - Alternative data structures & code variants

Optimizations Available in the Initial Release

- Optimizations for SpMV (**bold** → **heuristics**)
  - **Register blocking (RB)**: up to **4x** over CSR
  - Variable block splitting: 2.1x over CSR, 1.8x over RB
  - Diagonals: 2x over CSR
  - Reordering to create dense structure + splitting: 2x over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x over RB
  - **Cache blocking**: **3x** over CSR
  - **Multiple vectors (SpMM)**: **7x** over CSR
  - And combinations…
- Sparse triangular solve
  - Hybrid sparse/dense data structure: **1.8x** over CSR
- Higher-level kernels
  - **AA^T*x**, A^TA*x: **4x** over CSR, 1.8x over RB
  - A^2*x: 2x over CSR, 1.5x over RB

Current and Future Work

- Pre-release and docs available at **bebop.cs.berkeley.edu/oski**
  - Fortran wrappers in progress
  - Comments on interface welcome!
- Future work
  - PETSc integration
  - Port to additional architectures
    - Vectors
    - SMPs
  - Additional heuristics
    - Buttari, et al. (2005)

The End

(Extra slides follow)

Installation

- ./configure [options] [detect system info]
  - --with-blas={<lib>, no, yes}
  - --with-papi={<lib>, no, yes}
  - --with-index-type={int, long, <C-type>}
  - --with-value-type={single, double, complex, doublecomplex}
  - ...
- make [build lib & run off-line benchmarks]
- make install
- make check [optional testing]

Implementation

- Uses preprocessor to generate different integer/value type combinations from single set of sources
- Matrix type modules
  - Each matrix type is its own dynamically loaded module
  - Parameterized by scalar type, e.g., CSR<int, double>
  - Types "registered" at run-time
  - Module interface includes kernels, conversion, ...
- Kernels
  - Dispatch based on matrix type
    - Each type implements SpMV + any subset of other 4 kernels
  - Default implementations if matrix type does not implement particular kernel
  - Self-profiling: time, number of calls

What Happens at Tuning Time?

- Available information
  - Hints about matrix structure
  - Workload hints from user (# of calls to each kernel w/ particular options)
  - Trace: Observed calls and execution time
  - Time to stream through matrix (at matrix creation time)
- Tuning procedure
  - Estimate a “tuning budget” from trace & workload hints
    - (fraction) * MAX(workload “time”, trace time)
  - WHILE (time left for tuning) & (not tuned) DO
    - Get and try a heuristic
- Currently does not re-tune

Heuristic Models

- Each in its own dynamically loadable module
- Module interface to a heuristic
  - IsApplicable(<tuning info, e.g., matrix, trace, hints>);
  - <estimated time> = GetEstimatedCost(...);
  - <results> = Evaluate(...);
  - Transform(matrix, results);

Requires High-Resolution Timers

- Inline assembly cycle counter readers for most platforms
  - Adapted from FFTW-3.0 (MIT license)
  - Includes x86-32, x86-64, IA-64, Sun, PowerPC, PA-RISC
  - Also wraps around PAPI if available (configure-time)

Documentation & Testing

- Doxygen for API
- Large test suite
  - ~30k line matrix multiply test program tries {precision} x {pattern} x {0,1-based inds} x {op(A)} x {x-orient} x {y-orient}

Optimizations in the Initial OSKI Release

- Fully automatic heuristics for
  - Sparse matrix-vector multiply
    - Register-level blocking
    - Register-level blocking + symmetry + multiple vectors
    - Cache-level blocking
  - Sparse triangular solve with register-level blocking and “switch-to-dense” optimization
  - Sparse A^TA*x with register-level blocking
- User may select other optimizations manually
  - Diagonal storage optimizations, reordering, splitting; tiled matrix powers kernel (A^k*x)
  - All available in dynamic libraries
  - Accessible via high-level embedded script language