

Understanding and Optimizing the Performance of Scientific Software

(Developing HPC Software in an Architecture Zoo)

Boyana Norris

Assistant Professor

http://ix.cs.uoregon.edu/~norris

norris@cs.uoregon.edu

UNIVERSITY OF OREGON

Acknowledgments

- Funding:
  - DOE Scientific Discovery through Advanced Computing (SciDAC) program
  - National Science Foundation

- Collaborators:
  - Dr. S. H. K. Narayanan, P. Hovland, Azamat Mametjanov, Argonne National Laboratory
  - Dr. Albert Hartono, Reservoir Labs
  - Prof. P. (Saday) Sadayappan, Ohio State University
  - Prof. William Gropp, University of Illinois at Urbana-Champaign
  - Prof. Elizabeth Jessup, University of Colorado at Boulder
  - Prof. Jeremy Siek, Indiana University

Research summary

- Compiler techniques for source code analysis and transformation, e.g., for performance analysis
- Optimization of human and application performance
  - Embeddable domain-specific languages for performance portability and productivity
  - Code transformations, autotuning
  - Software taxonomies, adaptive algorithms and software

What does it involve?

- Many different areas of CS, math
  - Programming languages, compilers
  - Software engineering
  - Parallel computing
  - Numerical algorithms
  - Mathematical modeling (e.g., performance)
  - Data analysis (statistics, machine learning)
    - For modeling
    - For knowledge extraction (ontologies)
  - User interface design, databases, web applications
- And of course -- applications

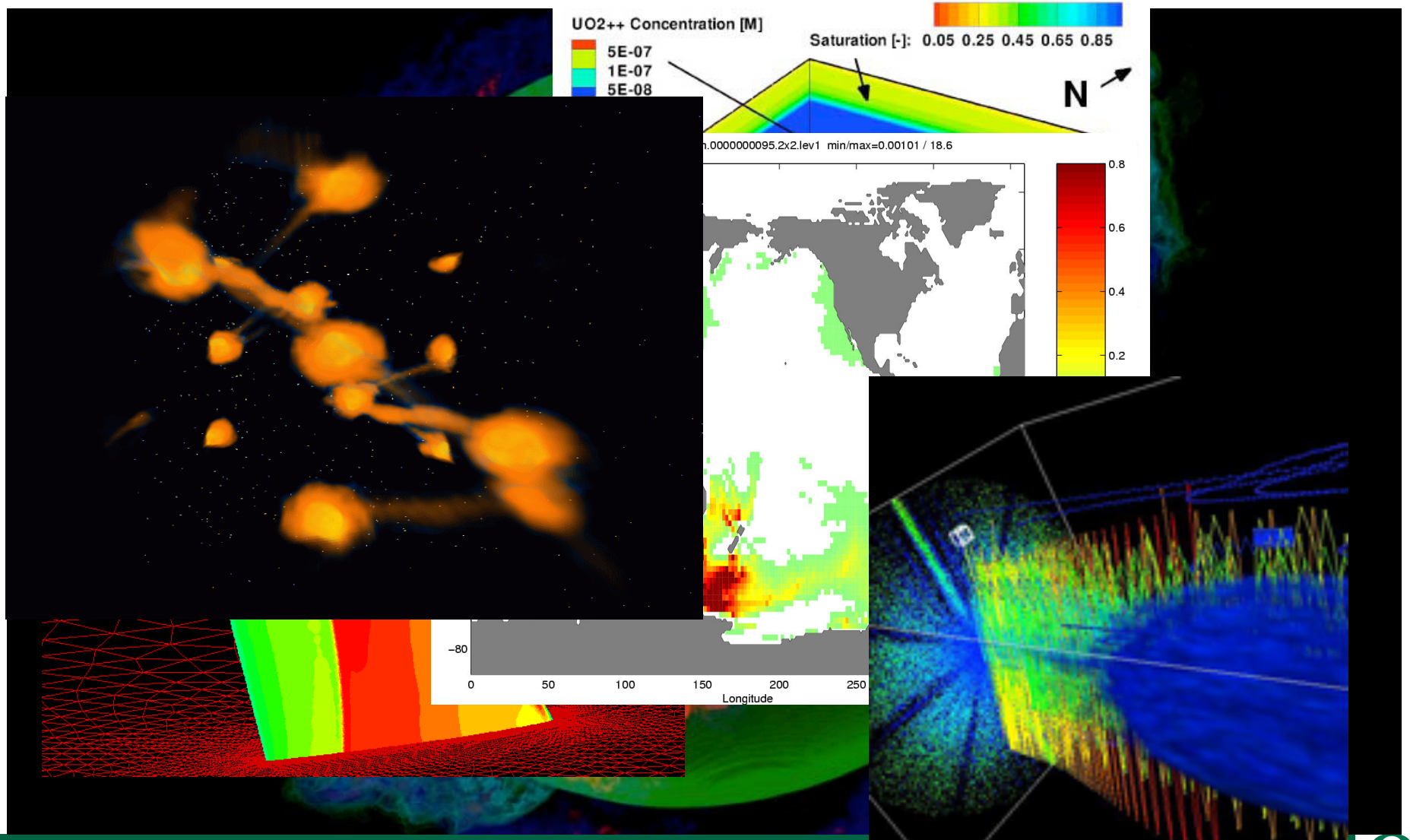
My background

- 2013 – present: Assistant Professor, Computer and Information Science, University of Oregon
- 2006 – 2013 Computer Scientist

  Mathematics and Computer Science Division, Argonne National Laboratory
- 2002 – 2006 Assistant Computer Scientist, Argonne National Laboratory
- 1999 – 2002 Postdoctoral researcher, Argonne
- 1995 – 1999 Ph.D. University of Illinois at Urbana-Champaign

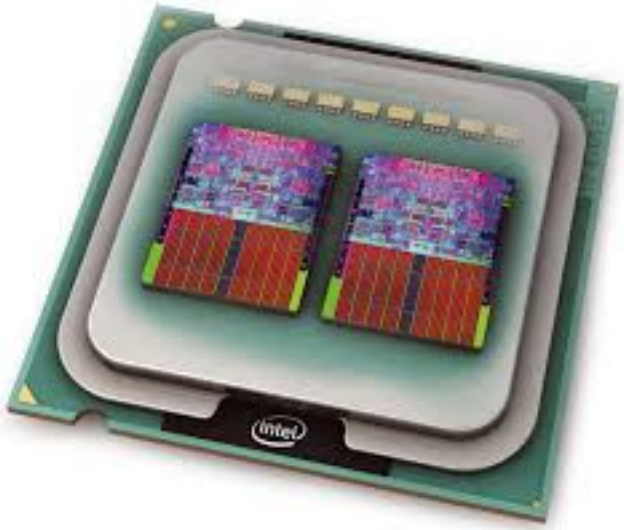
UNIVERSITY OF OREGON

Motivating applications



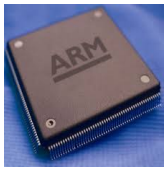
UNIVERSITY OF OREGON

Architecture zoo

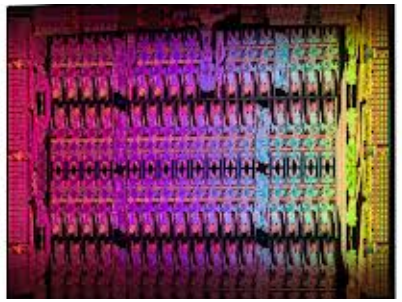


Multicore CPUs





Low power



Intel MIC





FPGAs



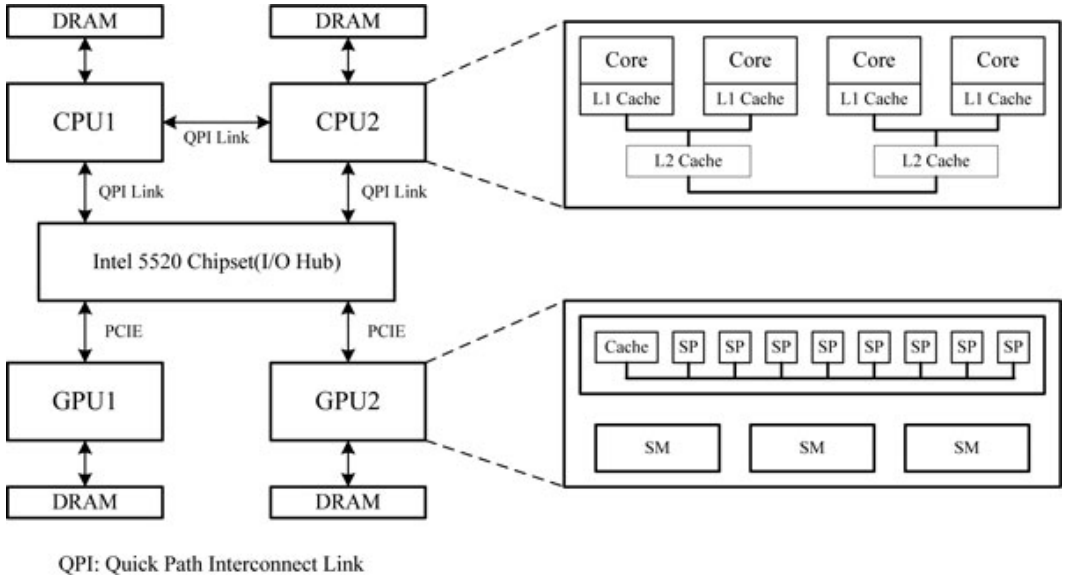
NVIDIA GPUs



AMD Radeon GPUs

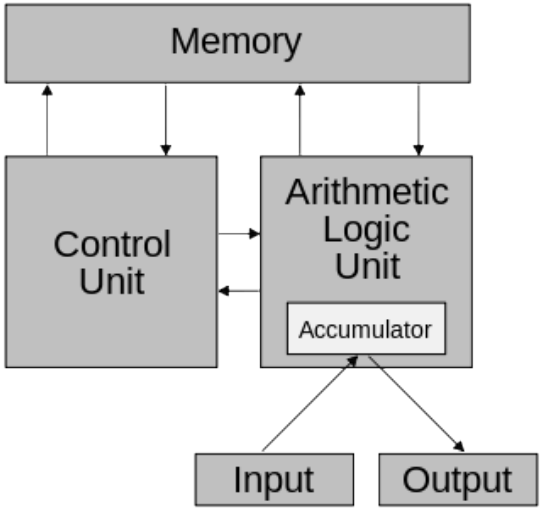
So what is the problem?

- Modern computers look like this:



QPI: Quick Path Interconnect Link

- Languages and supporting tools still mostly target this:



UNIVERSITY OF OREGON

Quick overview

- Part I: Understanding performance
  - Generating performance models from source code
- Part II: Automatically optimizing performance
  - Simple Example
  - Approach: Annotation-driven empirical tuning
  - Performance Experiments

Part I: Understanding performance

- Three main approaches to analyzing performance:
  - Empirical
  - Static (source or object code-based)
  - Analytical
- In this talk:
  - Focus on a hybrid static/empirical approach for generating performance models of computational kernels
  - Estimate *upper performance bounds*
    - Best achievable performance

What are performance bounds?

- Performance bounds give the upper limit in performance that can be expected for a given application on a given system
- Other approaches:
  - Fully algorithmic
    - Ignores (most) machine information
  - Theoretical peak of the machine
    - Based only on available floating point units
  - Fully dynamic with profiling
    - Profiling overheads, time consuming
- Our approach
  - Application signatures + Architectural Information => Bounds

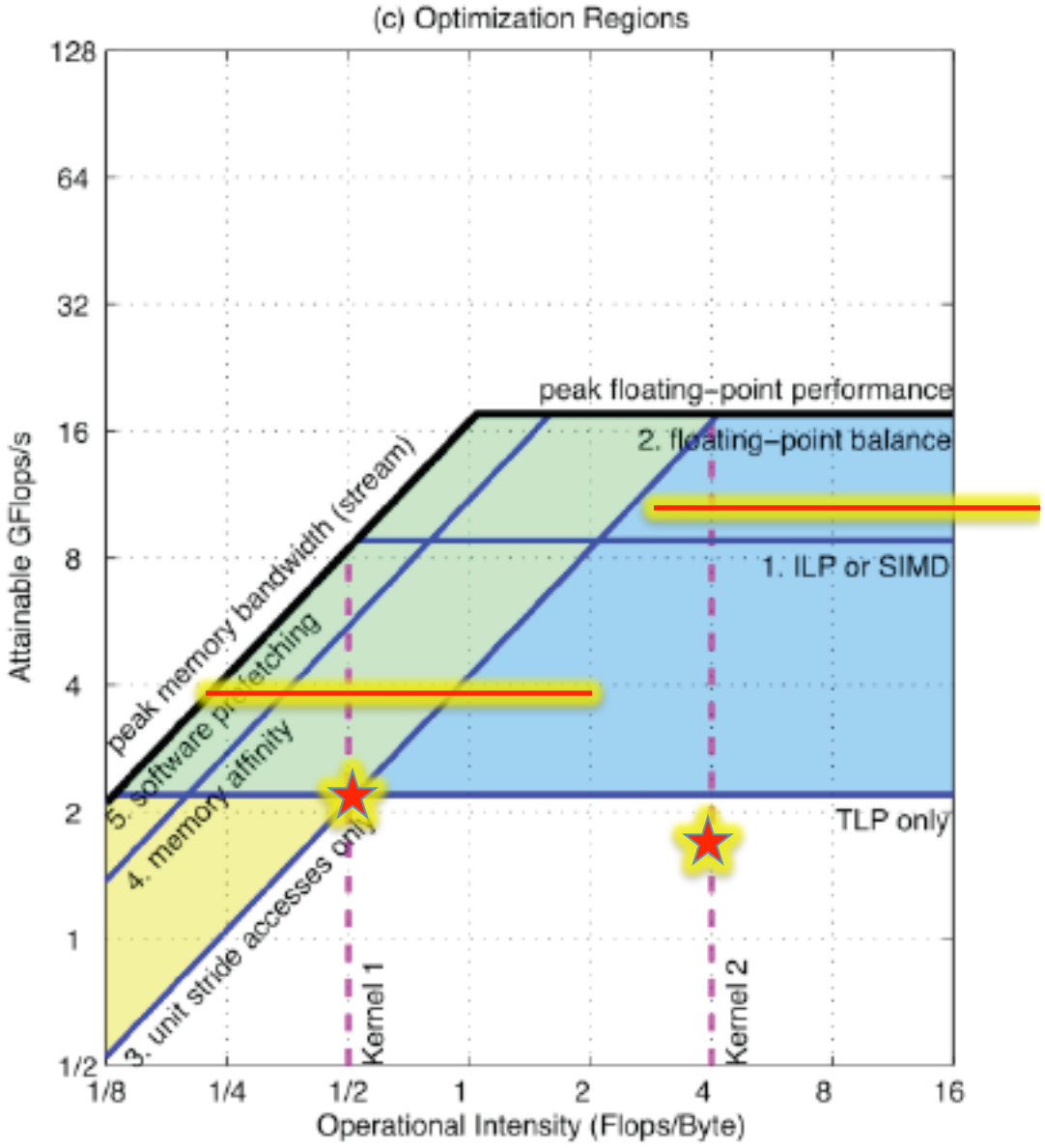
How does this help us?

- Determines the efficiency of the application
  - Efficiency = Observed Performance / Performance Bound
- Provides insight on how to improve performance
  - Low efficiency:
    - Re-engineer/tune the implementation
  - High efficiency but low performance:
    - Change algorithm or architecture
- Related work:
  - Gropp, Kaushik et al. provided the motivation for this work through similar manual analysis
  - Vuduc et al.: manual bounds estimation for sparse A^TAx computation kernel
  - Palm: new tools with similar goals

Example: Roofline models

- Shows the "region" of possible application performance
- Determines how optimizations affect application performance
- The performance space is determined through runtime means.

Source : S. Williams, A. Waterman and D. Patterson Communications of the ACM 2009



PBound

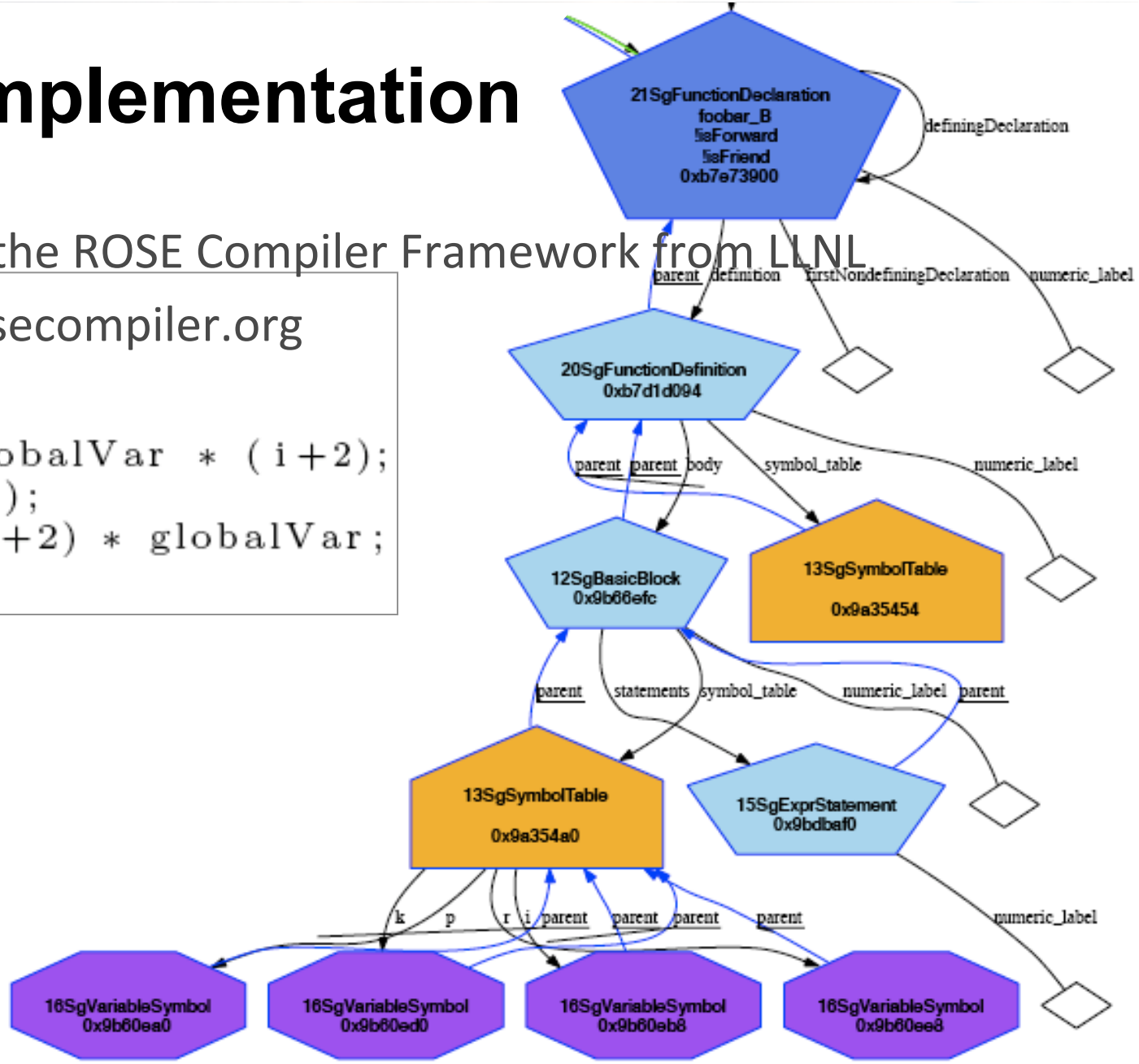
http://tinyurl.com/PBound

- A tool that generates performance bounds from C, C++, or Fortran source code
- Uses static (source code) analysis
  - Produces parameterized closed-form expressions expressing the computational and data load/store requirements of application kernels.
- Coupled with architectural information
  - Produces upper bounds on the performance of the application

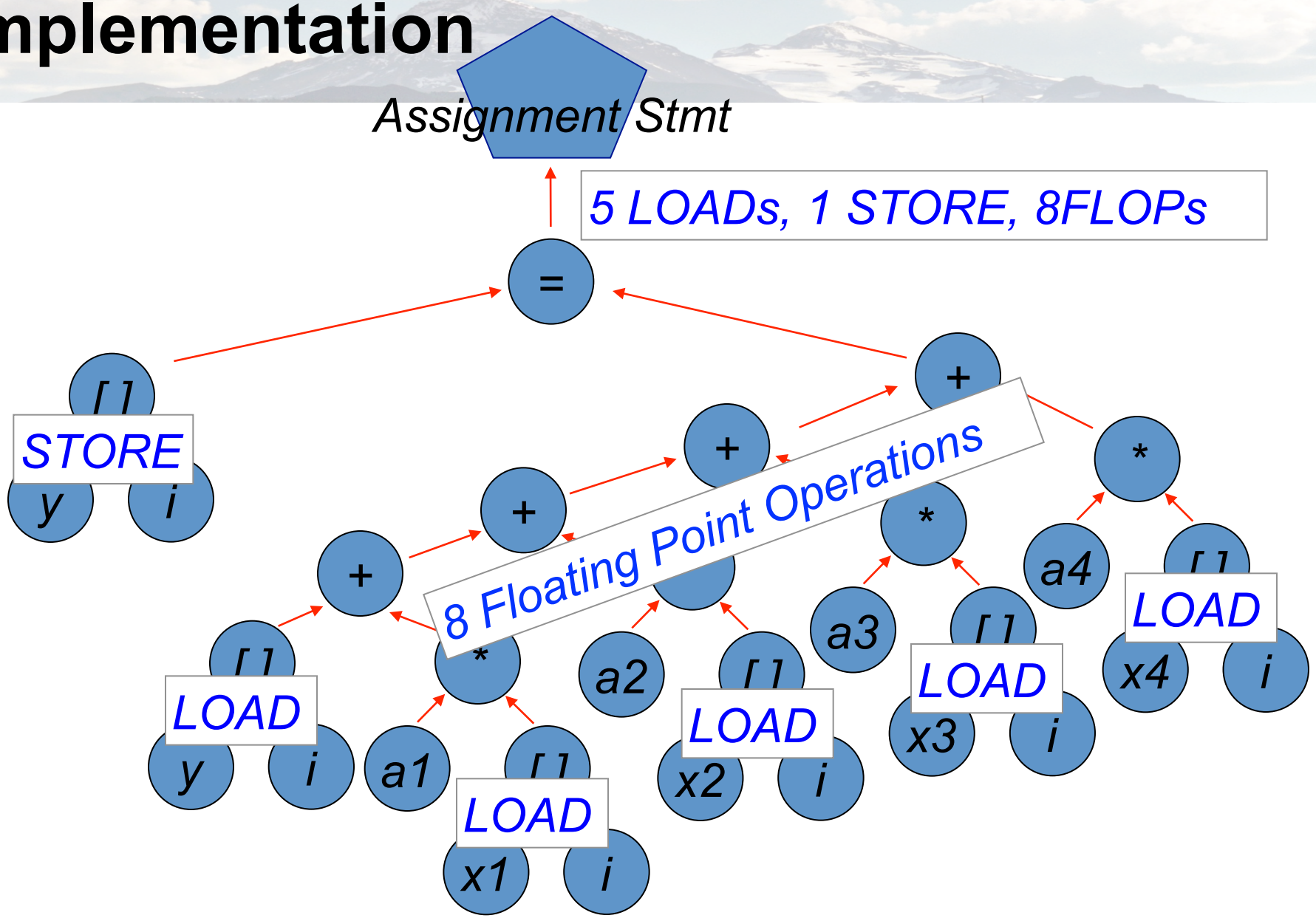
Pbound implementation

- Built on top of the ROSE Compiler Framework from LLNL
- http://www.rosecompiler.org

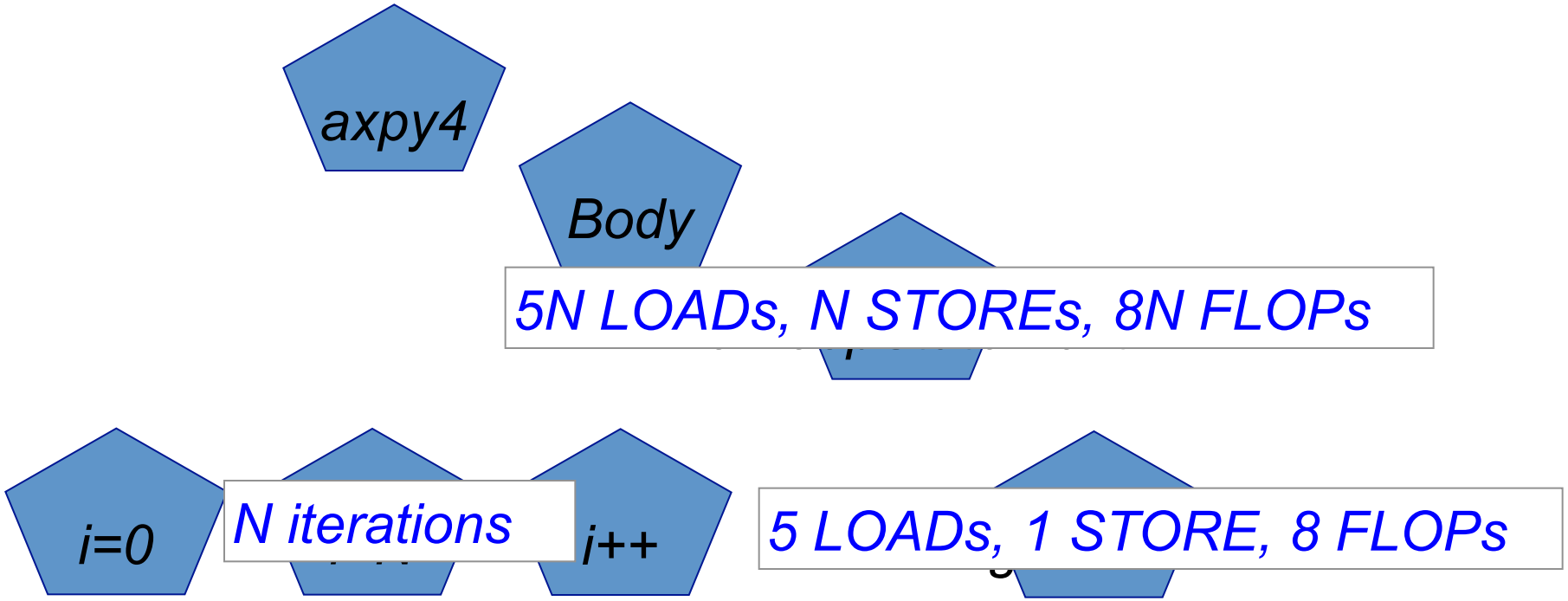
```
void foobar_B()
{
    int p;
    int i = 4;
    int k = globalVar * (i+2);
    p = foo (k);
    int r = (p+2) * globalVar;
}
```



Implementation



Implenentation (cont.)



Generated output (one version)

```
void axpy4(int n, double *y, double a1, double *x1,
  double a2, double *x2, double a3, double *x3, double
  a4, double *x4) {
    register int i;
    for (i=0; i<=n-1; i++)
      y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
}
```



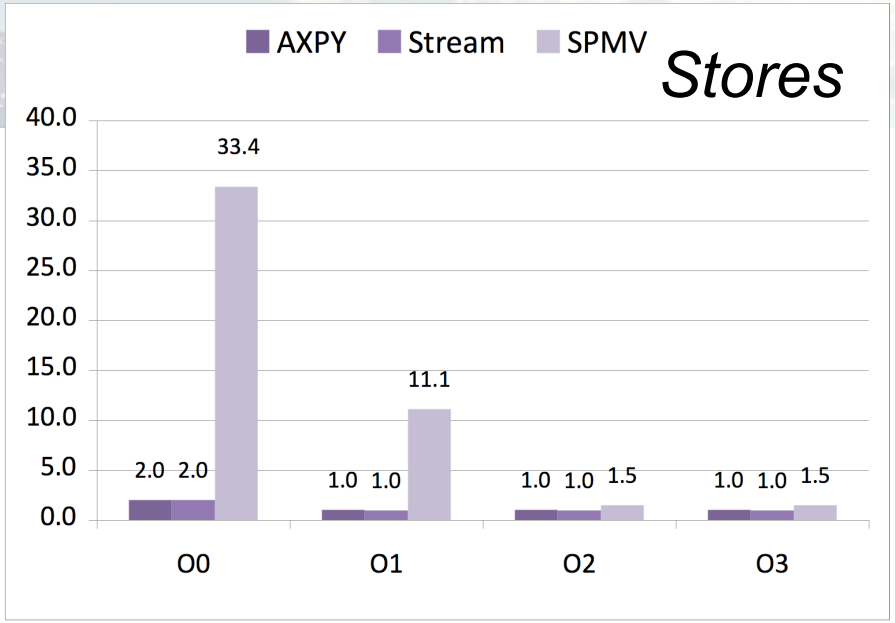
```
#include "pbound_list.h"
void axpy4(int n, double *y, double a1, double *x1,
  double a2, double *x2, double a3, double *x3, double
  a4, double *x4){
#ifdef pbound_log
pboundLogInsert("axpy.c@6@5",8,0,40 * ((n - 1) + 1) +
  32,      8 * ((n - 1) + 1),3 * ((n - 1) + 1) + 1,4 *
  ((n - 1) + 1));
 #endif
}
```

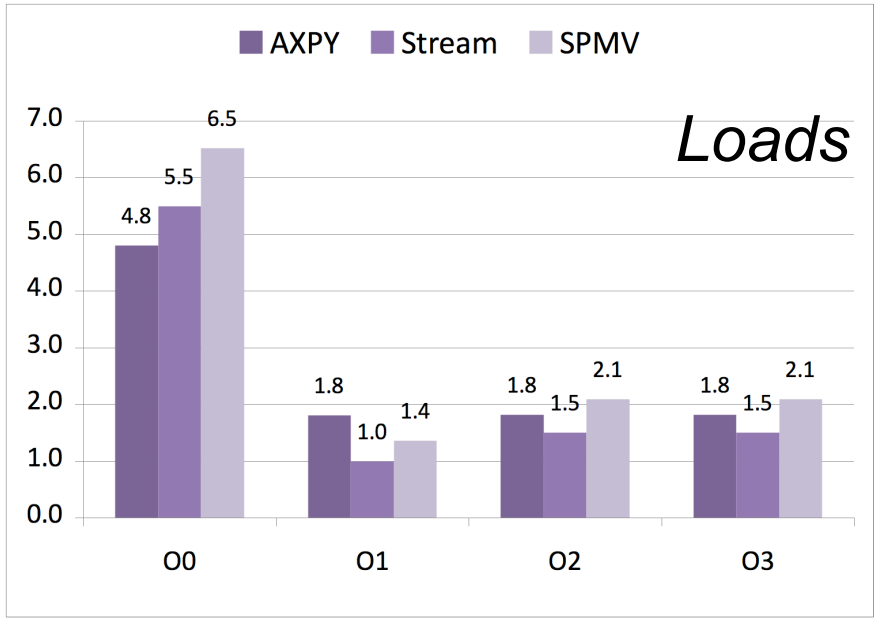
Architecture details

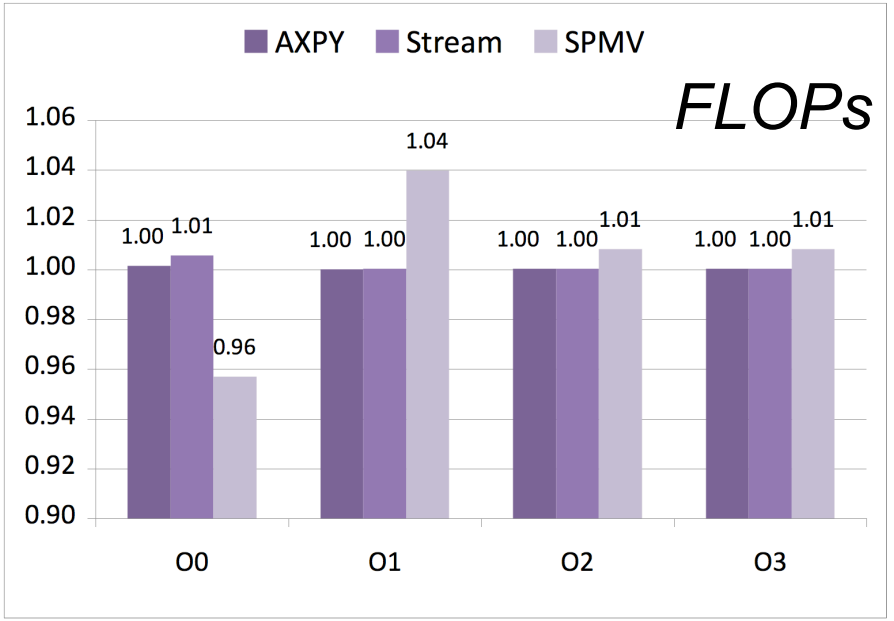
- Simple reuse distance analysis (to model behavior on cache-based architectures)
- SIMD (vector length, types of instructions)
- Fused Operations
  - fp_mul_add
  - fp_mul_sub
  - fp_div_add
  - fp_div_sub

Example results

- Comparison of static information with actual measurements
- Bars show ratio of observed value (measured with TAU) to predicted value







UNIVERSITY OF OREGON

Computing bounds

$$UpperBound = N * P_m$$

- N - Ratio of FLOPs to Bytes as determined by PBound
- P_m - Peak Reads Bandwidth determined by LMBench

	N (FLOPs/Byte)	UB GFLOP/s	UB/Peak
Axpy	1.60	6.01	53.70%
STREAM	1.00	3.76	33.56%
SPMV	0.67	2.51	22.38%

Where are static models used

- Evaluate the effectiveness of transformations (either manual or by tools, compilers)

- Empirical code tuning
  - Decide whether a transformation is likely to be an improvement or not without running the code
  - This helps make the search process more efficient

Part II: Optimizing performance

Motivation

- Abstractions provided by HPC programming languages or environments are not close enough to the human developer
- Lack of performance portability
- Compiler optimizations insufficient
- High performance often means decreased developer productivity
  - How many lines of code for “Hello, world!” in OpenCL?

Orio: Optimizing performance

http://brnorris03.github.io/Orio)

Motivation

- Abstractions provided by HPC programming languages or environments are not close enough to the human developer
- Lack of performance portability
- Compiler optimizations insufficient
- High performance often means decreased developer productivity
  - Q: How many lines of code for “Hello, world!” in OpenCL?
  - A: > 100

A Fast DGEMM (Sample)

```
      SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
     $                   BETA, C, LDC )
...
                  UISEC = ISEC-MOD( ISEC, 4 )
                  DO 390 J = JJ, JJ+UJSEC-1, 4
                     DO 360 I = II, II+UISEC-1, 4
                        F11 = DELTA*C( I,J )
                        F21 = DELTA*C( I+1,J )
                        F12 = DELTA*C( I,J+1 )
                        F22 = DELTA*C( I+1,J+1 )
                        F13 = DELTA*C( I,J+2 )
                        F23 = DELTA*C( I+1,J+2 )
                        F14 = DELTA*C( I,J+3 )
                        F24 = DELTA*C( I+1,J+3 )
                        F31 = DELTA*C( I+2,J )
                        F41 = DELTA*C( I+3,J )
                        F32 = DELTA*C( I+2,J+1 )
                        F42 = DELTA*C( I+3,J+1 )
                        F33 = DELTA*C( I+2,J+2 )
                        F43 = DELTA*C( I+3,J+2 )
                        F34 = DELTA*C( I+2,J+3 )
                        F44 = DELTA*C( I+3,J+3 )
                        DO 350 L = LL, LL+LSEC-1
                           F11 = F11 + T1( L-LL+1, I-II+1 )*
     $                                 T2( L-LL+1, J-JJ+1 )
                           F21 = F21 + T1( L-LL+1, I-II+2 )*
     $                                 T2( L-LL+1, J-JJ+1 )
                           F12 = F12 + T1( L-LL+1, I-II+1 )*
     $                                 T2( L-LL+1, J-JJ+2 )
                           F22 = F22 + T1( L-LL+1, I-II+2 )*
     $                                 T2( L-LL+1, J-JJ+2 )
                           F13 = F13 + T1( L-LL+1, I-II+1 )*
     $                                 T2( L-LL+1, J-JJ+3 )
                           F23 = F23 + T1( L-LL+1, I-II+2 )*
     $                                 T2( L-LL+1, J-JJ+3 )
                           F14 = F14 + T1( L-LL+1, I-II+1 )*
     $                                 T2( L-LL+1, J-JJ+4 )
                           F24 = F24 + T1( L-LL+1, I-II+2 )*
     $                                 T2( L-LL+1, J-JJ+4 )
                           F31 = F31 + T1( L-LL+1, I-II+3 )*
     $                                 T2( L-LL+1, J-JJ+1 )
                           F41 = F41 + T1( L-LL+1, I-II+4 )*
     $                                 T2( L-LL+1, J-JJ+1 )
                           F32 = F32 + T1( L-LL+1, I-II+3 )*
     $                                 T2( L-LL+1, J-JJ+2 )
                           F42 = F42 + T1( L-LL+1, I-II+4 )*
     $                                 T2( L-LL+1, J-JJ+2 )
                           F33 = F33 + T1( L-LL+1, I-II+3 )*
     $                                 T2( L-LL+1, J-JJ+3 )
                           F43 = F43 + T1( L-LL+1, I-II+4 )*
     $                                 T2( L-LL+1, J-JJ+3 )
                           F34 = F34 + T1( L-LL+1, I-II+3 )*
     $                                 T2( L-LL+1, J-JJ+4 )
                           F44 = F44 + T1( L-LL+1, I-II+4 )*
     $                                 T2( L-LL+1, J-JJ+4 )
  350                   CONTINUE
...
*     End of DGEMM.
*
      END
```

Why not just

```
do i=1,n
    do j=1,m
        c(i,j) = 0
        do k=1,p
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

Or even

```
C = A * B
```


UNIVERSITY OF OREGON

Approach

- Introduce annotations into existing code
  - Rewrite the annotated computation in one of the input languages accepted by Orio
  - Provide optimization *hints* as well as typical inputs to the computation being optimized
- Search the transformation space to produce an optimal or near-optimal implementation

Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
        double *x2, double a3, double *x3, double a4, double *x4)
{


  int i;









  for (i = 0; i < n; i++)
     y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];


}
```


UNIVERSITY OF OREGON

Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
        double *x2, double a3, double *x3, double a4, double *x4)
{
  /*@ begin PerfTuning(


  int i;
  /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/




  for (i = 0; i < n; i++)
     y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];

  /*@ end @*/
}
```


UNIVERSITY OF OREGON

Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
        double *x2, double a3, double *x3, double a4, double *x4)
{
  /*@ begin PerfTuning(

  )
  int i;
  /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
  /*@ begin Loop (
      transform Unroll(ufactor=UF, parallize=PAR, simd=SIMD_TYPE)
        for (i=0; I <= N-1 ; i++)
          y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
  ) @*/
  for (i = 0; i < n; i++)
     y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
  /*@ end @*/
  /*@ end @*/
}
```

Example (cont.)

```
spec axpy4_tune_spec {
 def build {
  arg build_command = 'mpixlc_r';
  arg batch_command =
   'qsub -n 128 -t 20 --env "OMP_NUM_THREADS=4"';
  arg status_command = 'qstat';
  arg num_procs = 128;
 }
 def performance_counter {
  arg method = 'basic timer';
  arg repetitions = 100;
 }
 def performance_params {
  param CFLAGS[] = ['-O0', '-O1','-O2',
        '-O3 -qstrict -qhot -qsmp=omp:noauto'];
  param UF[] = range(1,33);
  param PAR[] = [True,False];
  param SIMD_TYPE[] = ['none','xlc'];
  constraint simd_unroll_factor = (SIMD_TYPE=='none'
       or UNROLL_FAC_IN%2==0);
 }
# continued -> ...
```

```
def input_params {
  param N[] =
   [10,100,1000,10**4,10**5,
                10**6,10**7];
 }

def input_vars {
  decl dynamic double y[N] = 0;
  decl dynamic double x1[N] =
   random;
  decl double a1 = random;
  decl double a2 = random;
  # ... omitted ...
 }
 def search {
  arg algorithm = 'Simplex';
  arg time_limit = 20;
 }
}
```

Generated Code Fragment (full code > 100 lines)

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
        double *x2, double a3, double *x3, double a4, double *x4)
{
  int i;
  /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
  #pragma disjoint (*x1, *x2, *x3, *x4, *y)
  if ((((int)(x1)|(int)(x2)|(int)(x3)|(int)(x4)|(int)(y)) & 0xF) == 0) {
    __alignx(16,x1);
    __alignx(16,x2);
    __alignx(16,x3);
    __alignx(16,x4);
    __alignx(16,y);

    /*@ begin Loop (
      transform Unroll(ufactor=UF, parallize=PAR)
        for (i=0; i<=N-1; i++)
          y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    ) @*/
    #if ORIGLOOP
      for (i = 0; i < n; i++)
        y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    #else
      {
```

```
for ((i = 0); (i <= (n - 1) - 3); (i += 4))  {
      {
        double _Complex _i_541, _i_542, _i_543, _i_544, _i_545, _i_555, _i_556,
_i_664, _i_665;
        _i_541 = __lfpd(&y[i]);
        _i_542 = __lfpd(&x4[i]);
        _i_543 = __fxcpmadd(_i_541, _i_542, a4);
        _i_544 = __lfpd(&x3[i]);
        _i_545 = __fxcpmadd(_i_543, _i_544, a3);
        _i_555 = __lfpd(&x2[i]);
        _i_556 = __fxcpmadd(_i_545, _i_555, a2);
        _i_664 = __lfpd(&x1[i]);
        _i_665 = __fxcpmadd(_i_556, _i_664, a1);
        __stfpd(&y[i], _i_665);
      }
      {
        double _Complex _i_541, _i_542, _i_543, _i_544, _i_545, _i_555, _i_556,
_i_664, _i_665;
        _i_541 = __lfpd(&y[(i + 2)]);
        _i_542 = __lfpd(&x4[(i + 2)]);
        _i_543 = __fxcpmadd(_i_541, _i_542, a4);
        _i_544 = __lfpd(&x3[(i + 2)]);
        _i_545 = __fxcpmadd(_i_543, _i_544, a3);
        _i_555 = __lfpd(&x2[(i + 2)]);
        _i_556 = __fxcpmadd(_i_545, _i_555, a2);
        _i_664 = __lfpd(&x1[(i + 2)]);
        _i_665 = __fxcpmadd(_i_556, _i_664, a1);
        __stfpd(&y[(i + 2)], _i_665);
      }
```


UNIVERSITY OF OREGON

Simple Example

```
void axpy_4(int n, double *y, double a1, double *x1, double a2,
        double *x2, double a3, double *x3, double a4, double *x4)
{
  /*@ begin PerfTuning(
      import spec axpy4_tune_spec;
  ) @*/
  int i;
  /*@ begin Align (x1[],x2[],x3[],x4[],y[]) @*/
  /*@ begin Loop (
      transform Unroll(ufactor=UF, parallize=PAR, simd=SIMD_TYPE)
        for (i=0; I <= N-1 ; i++)
          y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
  ) @*/
  for (i = 0; i < n; i++)    /* Original code */
     y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
  /*@ end @*/
  /*@ end @*/
}
```


UNIVERSITY OF OREGON

More Complex Annotation Example (Matrix-Matrix Product)

```
/*@ begin Loop(
  transform Composite(
    tile = [('i',T1_I,'ii'),('j',T1_J,'jj'),('k',T1_K,'kk'),
            (('ii','i'),T2_I,'iii'),(('jj','j'),T2_J,'jjj'),(('kk','k'),T2_K,'kkk')],
    permut = [PERMUTS],
    arrcopy = [(ACOPY_A,'A[i][k]',[(T1_I if T1_I>1 else T2_I),(T1_K if T1_K>1 else T2_K)],'_copy'),
               (ACOPY_B,'B[k][j]',[(T1_K if T1_K>1 else T2_K),(T1_J if T1_J>1 else T2_J)],'_copy'),
               (ACOPY_C,'C[i][j]',[(T1_I if T1_I>1 else T2_I),(T1_J if T1_J>1 else T2_J)],'_copy')],
    unrolljam = [('k',U_K),('j',U_J),('i',U_I)],
    scalarreplace = (SCREP, 'double', 'scv_'),
    vector = (VEC, ['ivdep','vector always']),
    openmp = (OMP, 'omp parallel for private(iii,jjj,kkk,ii,jj,kk,i,j,k,A_copy,B_copy,C_copy)')
  )

  for(i=0; i<=M-1; i++)
    for(j=0; j<=N-1; j++)
      for(k=0; k<=K-1; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];

) @*/
```


UNIVERSITY OF OREGON

Another input language example

```
GEMVER
in
  A : column matrix,
  u1 : vector, u2 : vector,
  v1 : vector, v2 : vector,
  a : scalar, b : scalar,
  y : vector, z : vector
out
  B : column matrix,
  x : vector, w : vector
{
  B = A + u1 * v1'
            + u2 * v2'
  x = b * (B' * y) + z
  w = a * (B * x)
}
```

- BTO: Build-to-order linear algebra compiler[1]
  - Annotation language uses slightly extended MATLAB syntax
  - Use BTO compiler to generate C code
  - Perform analysis of the C code
  - Tune with Orio

[1]Jessup, E., Karlin, I., , Siek, J.:
Build to order linear algebra kernels. *In: Proceedings of the IEEE International Symposium on Parallel and Distributed (IPDPS 2008)*, IEEE (2008) 1–8.

Tunable Numerical Kernel Examples

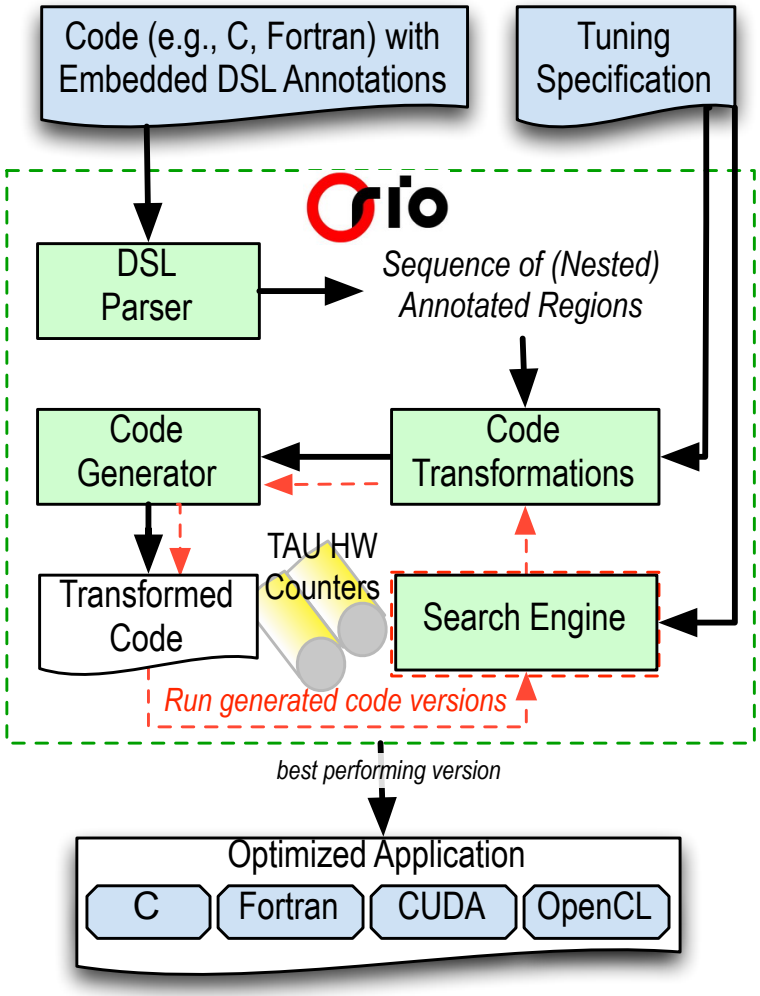
- For example, dense matrix-matrix product (DGEMM)
  - Example code from "Superscalar GEMM-based Level 3 BLAS", Gustavson et al. (next slide)

- BLAS-like operations for which there are no vendor-optimized libraries
  - Jacobian and Hessian accumulation code in AD-generated derivative computations
  - Tensor products
  - Composed linear algebra operations, e.g., $\boldsymbol{y} = a(\boldsymbol{A}\boldsymbol{x}) + b(\boldsymbol{B}\boldsymbol{x})$ where a and b are scalar variables, $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors, and $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices

- PETSc code for sparse matrix operations
  - Manual optimizations Include unrolling and use of registers
  - Code for diagonal matrix format is fast on cache-based systems but slow on vector systems
    - Too much code to rewrite by hand for new architectures

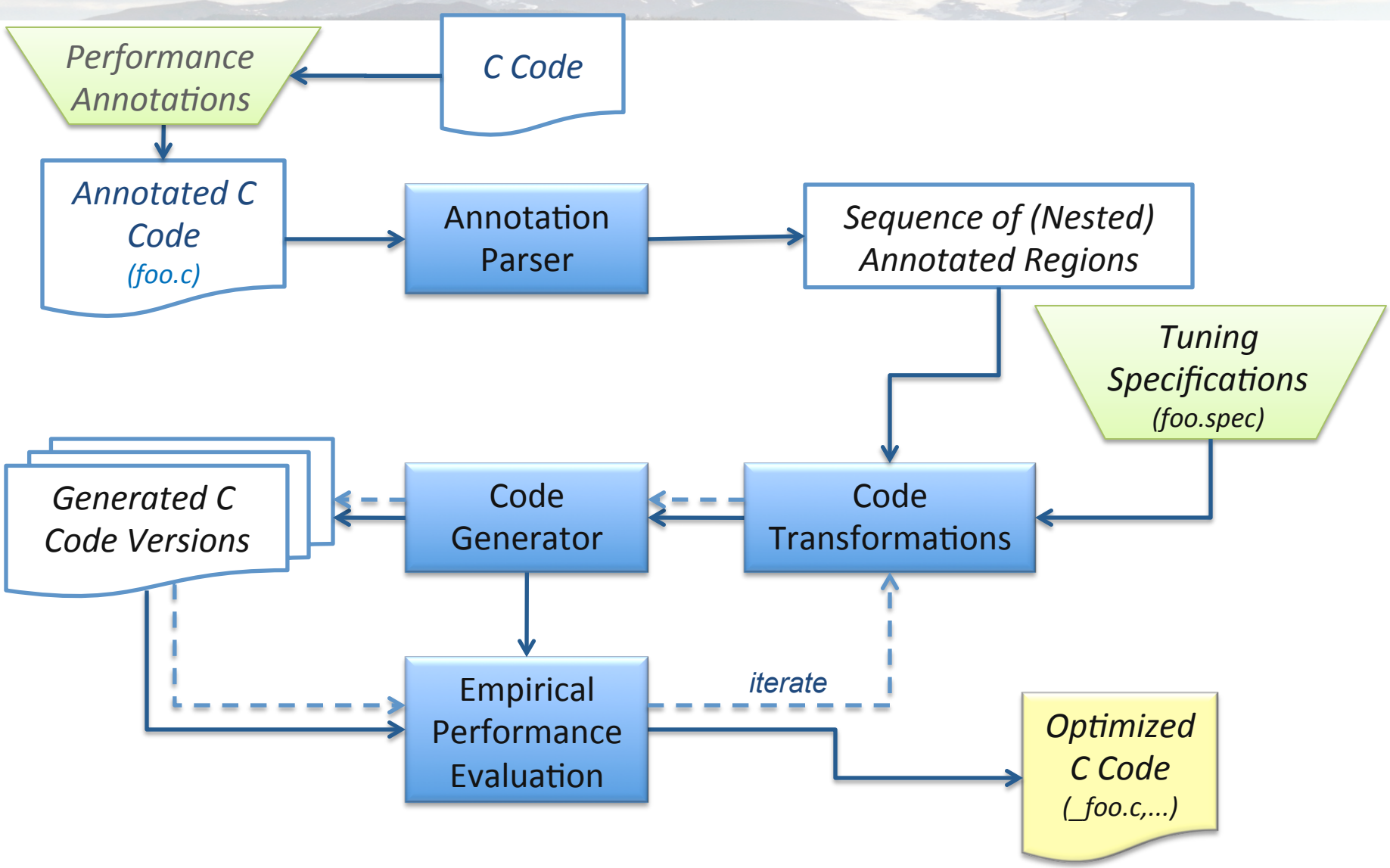
Orio

(http://brnorris03.github.io/Orio/)



```
orcc -spec=foo.spec -output=foo.c foo.c
```

Orio Workflow



UNIVERSITY OF OREGON

Search Engine

- Inputs:
  - Transformation specification
  - Performance experiment definition
    - Program inputs
    - Performance parameters
    - Constraints
    - Search method
    - Build and execution environment description
- Current search strategies:
  - Exhaustive
  - Random
  - Two variants of a Nelder-Mead Simplex algorithm with a time limit constraint
  - Simulated annealing
  - Ant colony optimization (in progress)

Autotuning Applications

- Orio transforms relatively small portions of library or application implementations
- Many applications spend most of their time in a few key kernels
- Kernel-level autotuning not guaranteed but likely beneficial for whole-application performance
  - Specialization of library implementations for specific applications (i.e., input parameter ranges)
  - Ability to tune for different control flow scenarios

Example: Really automatic autotuning of PETSc on NVIDIA GPUs

Enable autotuning by configuring PETSc with Orio as a compiler wrapper, e.g.

```
./configure --with-cc="orcc -r mpicc" --with-cxx="orcc -r mpicxx"
--with-fc=0 --download-f2cblaslapack
```

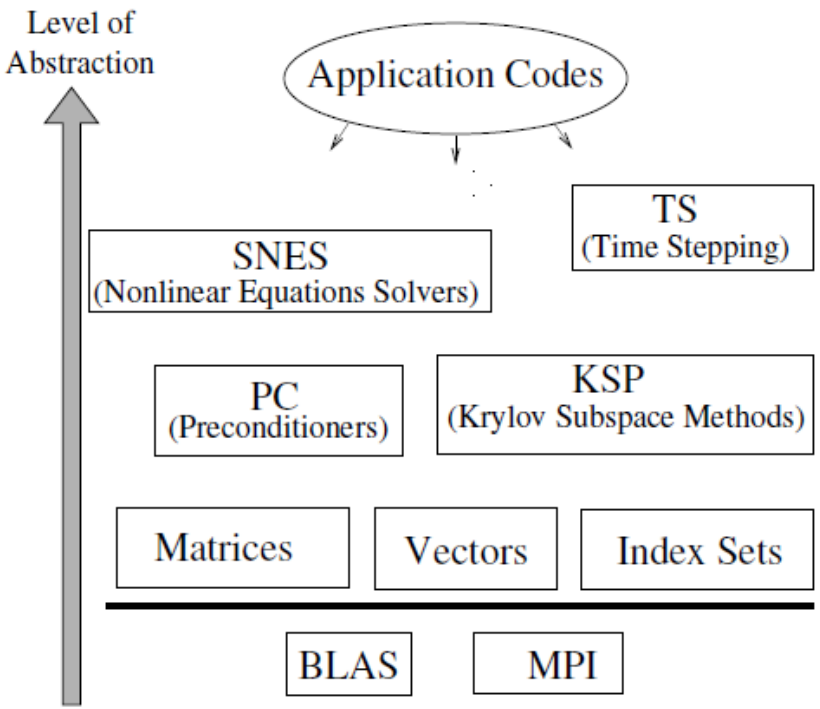
During the installation, pragma-annotated code is empirically autotuned, for example in the function computing *w = y + αx* (*w, y, x*: vectors, *α:* scalar)

```
PetscErrorCode VecWAXPY_Seq(Vec win, PetscScalar alpha,Vec xin,Vec
yin) {
  ...
#pragma Orio PerfTuning(import spec tune/waxpy;)
  ...
#pragma Orio Loop(transform Unroll(ufactor=UF, parallelize=PAR))
    for (i=0; i<=n-1; i++) ww[i] = yy[i] + xx[i];
  ...
#pragma Orio Loop(transform Unroll(ufactor=UF, parallelize=PAR))
    for (i=0; i<=n-1; i++) ww[i] = yy[i] + oalpha * xx[i];
  ...
}
```

Autotuning in PETSc

- Suite of data structures and routines for solutions of applications modeled by PDEs.
- Widely used parallel numerical library.
- Uses MPI internally.

- **Goal for autotuning: exploit known structure as much as possible**

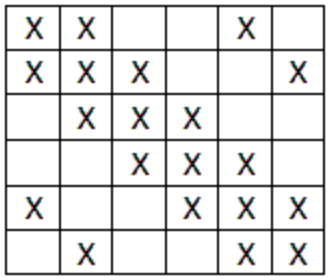


Organization of the PETSc Libraries

Structured Grid Computations

- Key Computational pattern in Scientific Computing.
- Applications in reservoir simulations, geoscience, biology etc.
- Stencil function – crux of structured grid computations.
  - Example: $A_{temp}[\,i\,][\,j\,] = (A[\,i\,][\,j\,] + A[\,i-1\,][\,j\,] + A[\,i+1\,][\,j\,] + A[\,i\,][\,j-1\,] + A[\,i\,][\,j+1\,]\,) / 5$
  - Explicit methods apply the stencil operator to every point in the grid explicitly.
  - Implicit methods form a sparse linear system

    Ax = B and solve using iterative linear solvers.

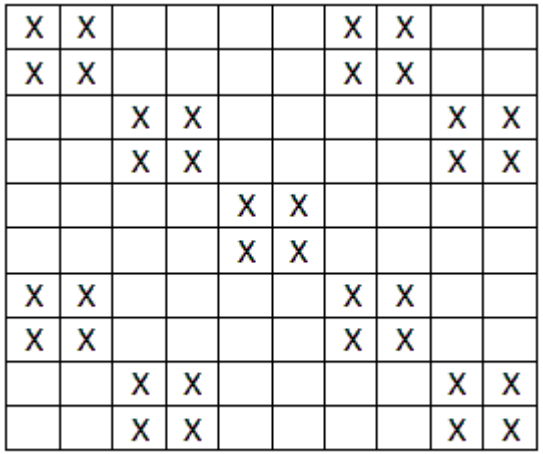


Sparsity Pattern

UNIVERSITY OF OREGON

Newton-Krylov solvers for structured grid problems

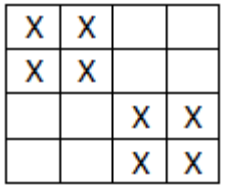
- Vector quantities at grid points and matrices as coefficients that relate the grid points.
- For grid size *m x n* the sparse matrix dimensions are (m*n*dof) X (m*n*dof).



Sparsity pattern for dof = 2

- The storage format for the sparse matrix has significant impact on memory efficiency and performance of computational kernels.

AIJ and Blocked AIJ Formats

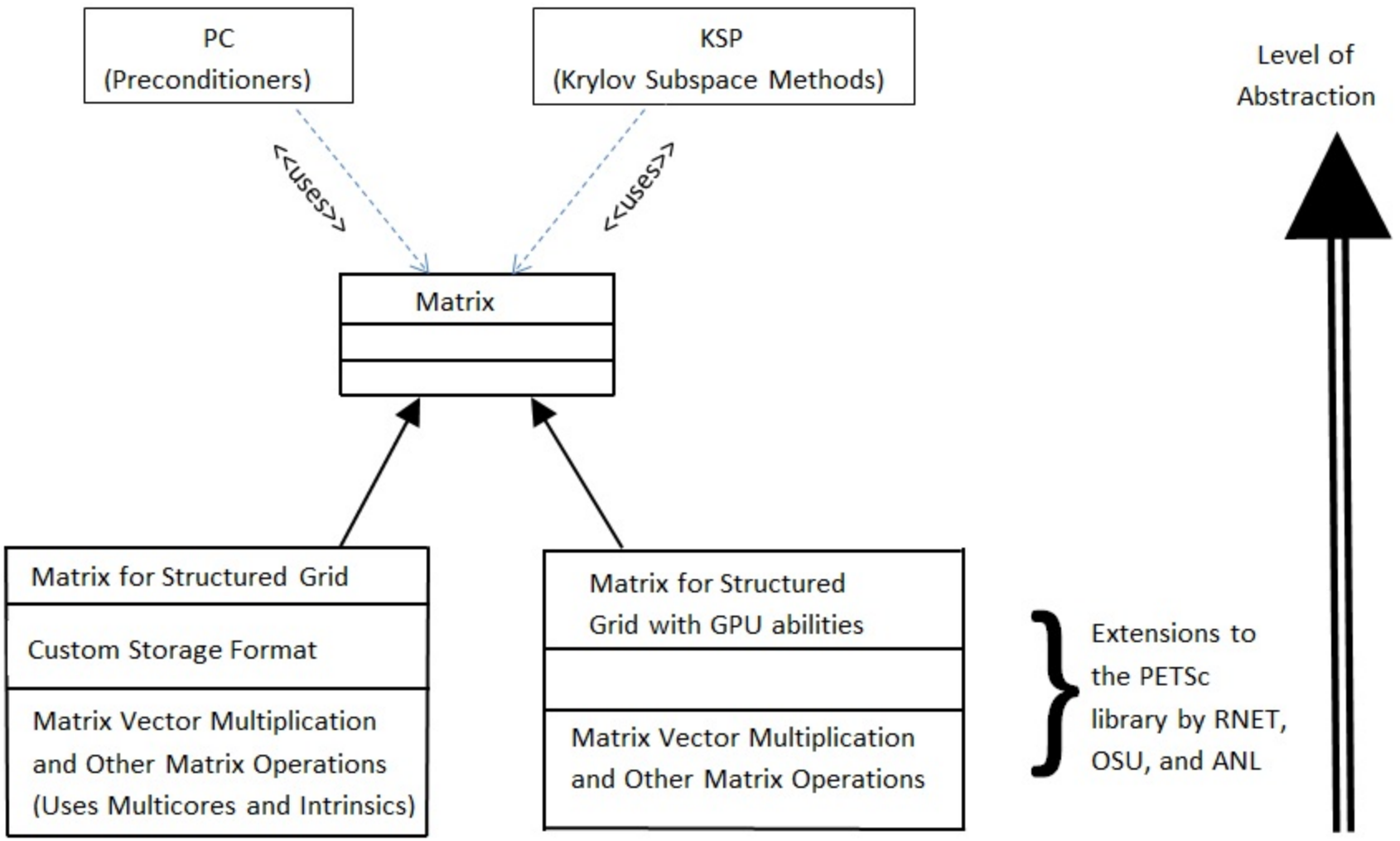


- AIJ Format:
  - A: X X X X X X X X
  - I: 1 1 2 2 3 3 4 4
  - J: 1 3 5 7 9
- Matrix Vector Multiplication:

```
for (i = 0;  i <  n;  ++i)
 for(j = rows[i];  j < rows[i+1];  ++j)
     y[i] += vals[j] * x[cols[j]];     //indirection
```

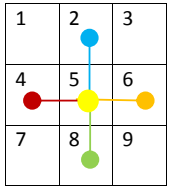
- The indirection renders most compiler optimizations impossible.

UNIVERSITY OF OREGON



UNIVERSITY OF OREGON

Sparse matrix algebra, stencil-based computations



(a)

Grid Element	1		2		3		4		5		6		7		8		9	
1	57	58	93	94			125	126										
	59	60	95	96			127	128										
2	25	26	61	62	97	98			129	130								
	27	28	63	64	99	100			131	132								
3			29	30	65	66	101	102			133	134						
			31	32	67	68	103	104			135	136						
4	1	2			33	34	69	70	105	106			137	138				
	3	4			35	36	71	72	107	108			139	140				
5			5	6			37	38	73	74	109	110			141	142		
			7	8			39	40	75	76	111	112			143	144		
6					9	10			41	42	77	78	113	114			145	146
					11	12			43	44	79	80	115	116			147	148
7							13	14			45	46	81	82	117	118		
							15	16			47	48	83	84	119	120		
8									17	18			49	50	85	86	121	122
									19	20			51	52	87	88	123	124
9											21	22			53	54	89	90
											23	24			55	56	91	92

(b)

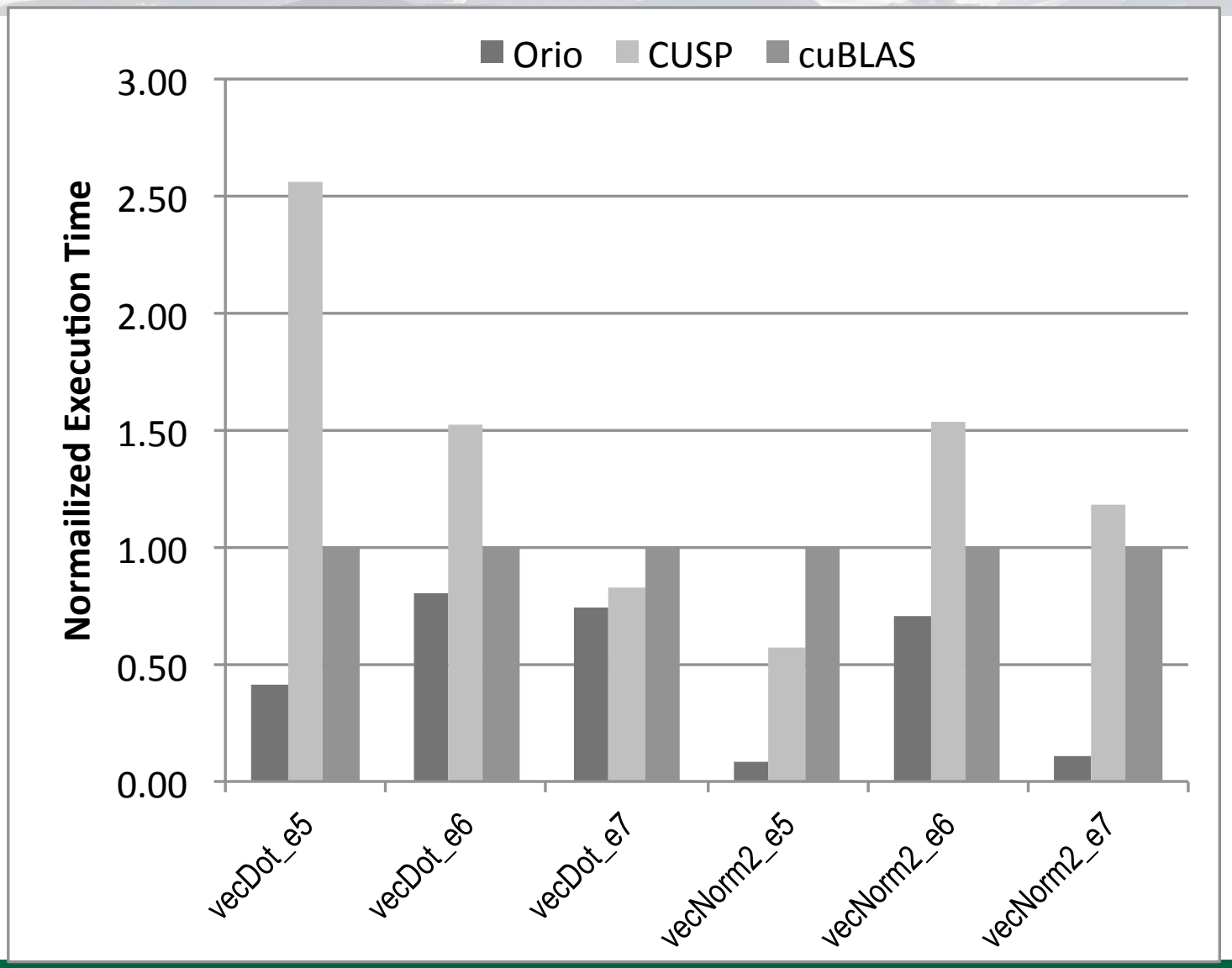
-3		-1		0		1		3	
00	00	00	00	57	58	93	94	125	126
00	00	00	00	59	60	95	96	127	128
00	00	25	26	61	62	97	98	129	130
00	00	27	28	63	64	99	100	131	132
00	00	29	30	65	66	101	102	133	134
00	00	31	32	67	68	103	104	135	136
1	2	33	34	69	70	105	106	137	138
3	4	35	36	71	72	107	108	139	140
5	6	37	38	73	74	109	110	141	142
7	8	39	40	75	76	111	112	143	144
9	10	41	42	77	78	113	114	145	146
11	12	43	44	79	80	115	116	147	148
13	14	45	46	81	82	117	118	00	00
15	16	47	48	83	84	119	120	00	00
17	18	49	50	85	86	121	122	00	00
19	20	51	52	87	88	123	124	00	00
21	22	53	54	89	90	00	00	00	00
23	24	55	56	91	92	00	00	00	00

(c)

UNIVERSITY OF OREGON

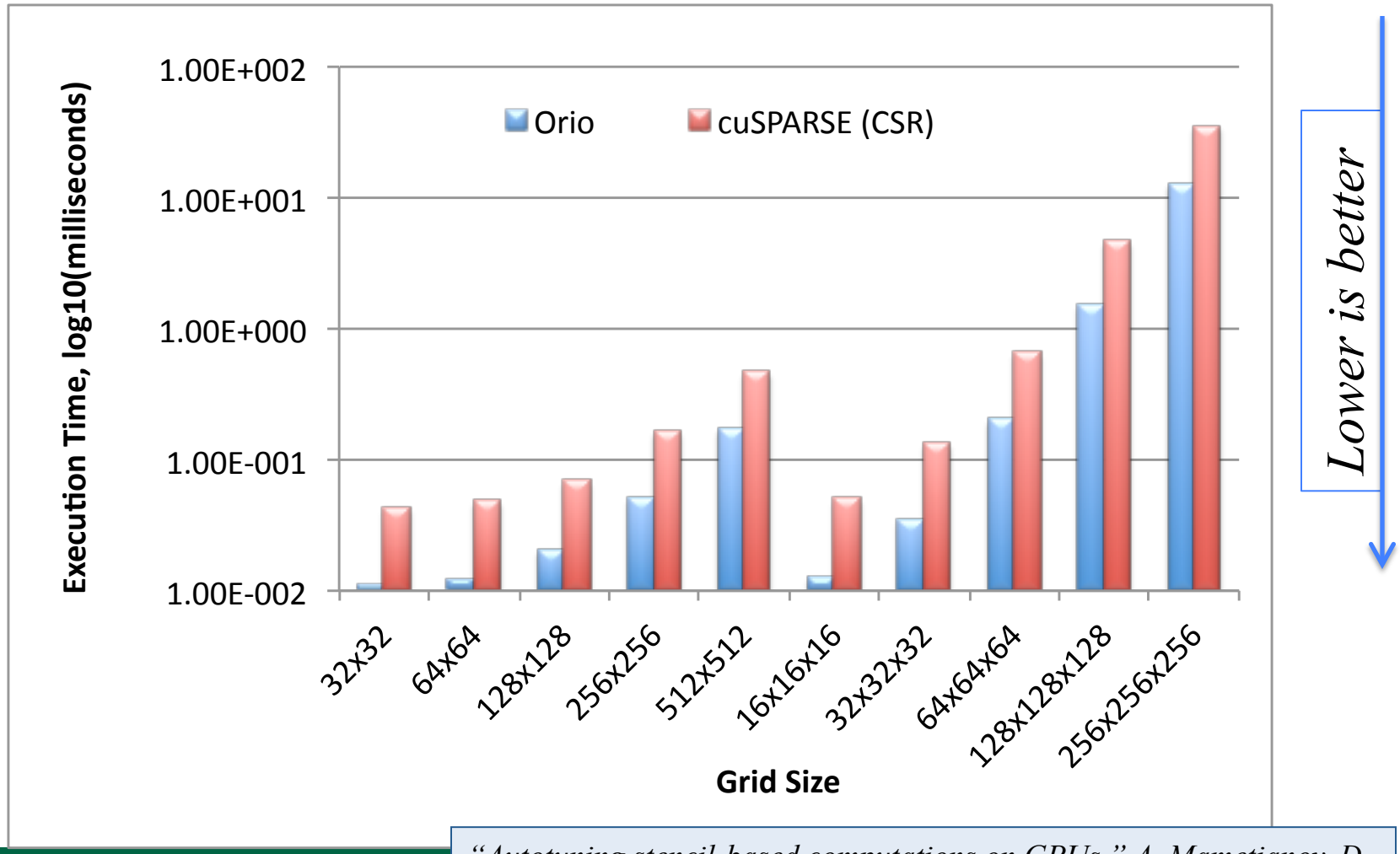
Reduction kernels

Intel Xeon (dual quad-core E5462 processors), 2.8GHz; GPU: NVIDIA Fermi C2070



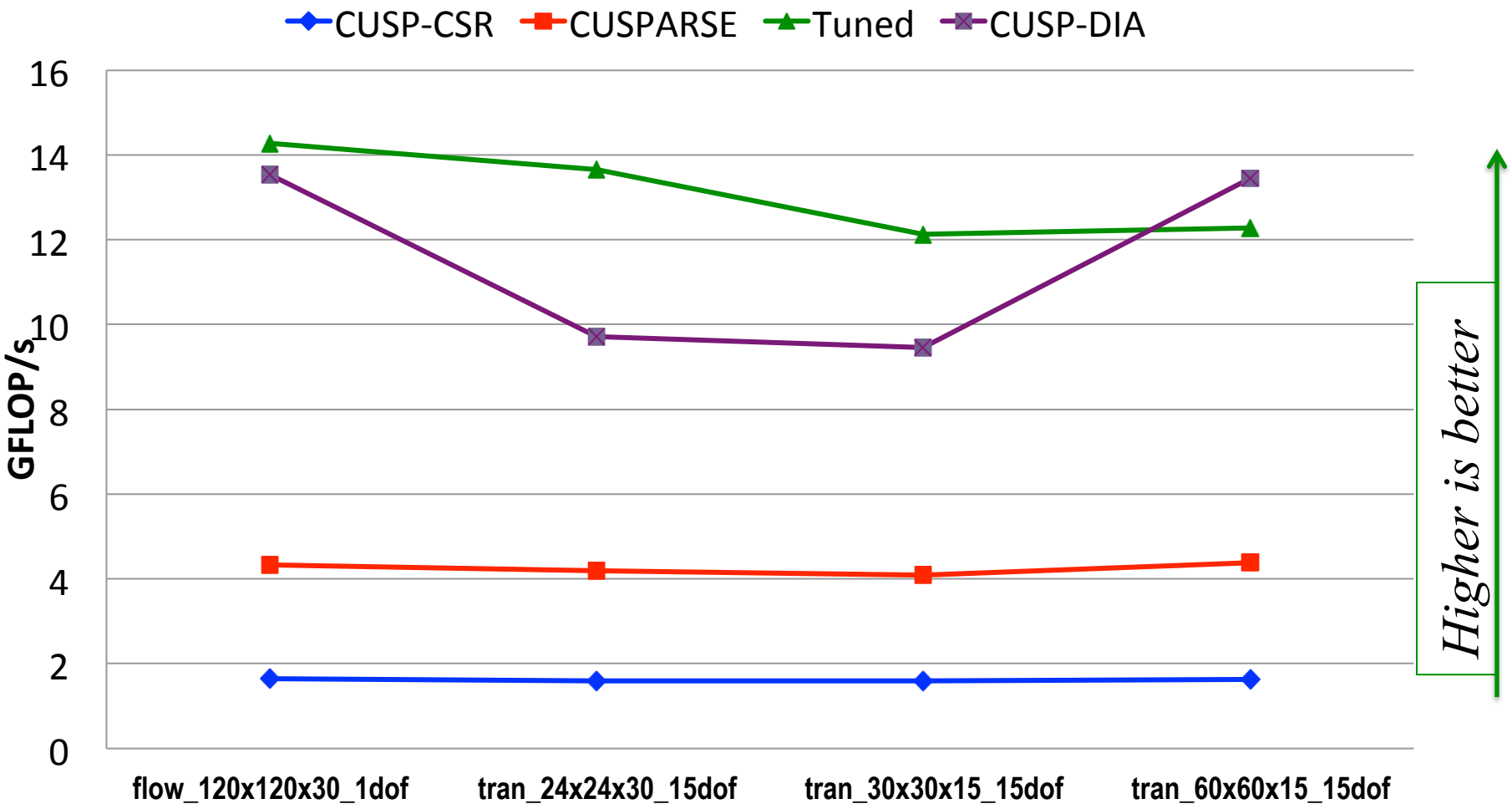
Example: Sparse matrix-vector product (7-point stencil) on a GPU

Intel Xeon (dual quad-core E5462 processors), 2.8GHz; GPU: NVIDIA Fermi C2070



"Autotuning stencil-based computations on GPUs." A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. Proceedings of IEEE Cluster 2012.

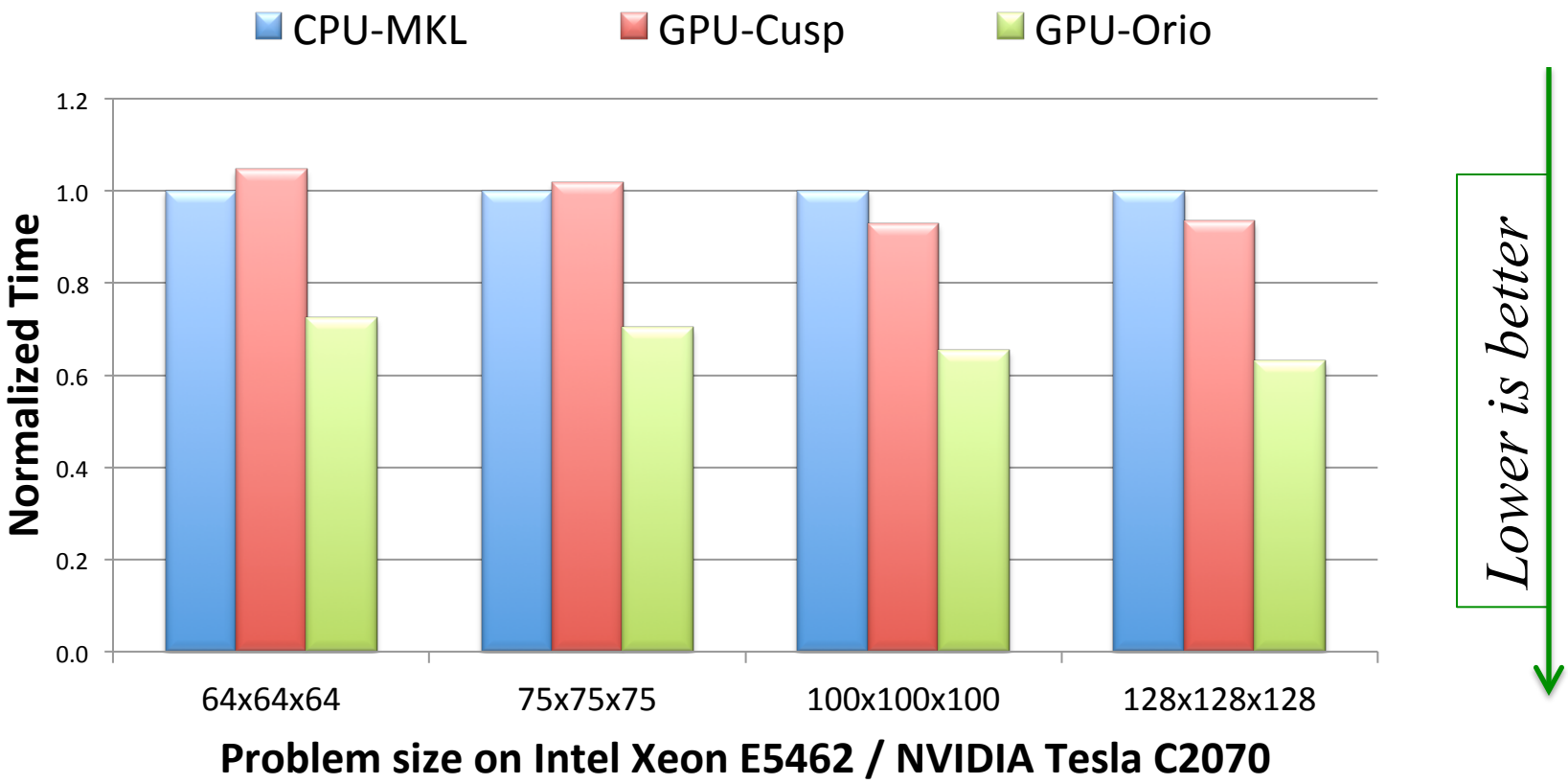
SpMV (PFLOTRAN matrices; NVIDIA C2070)



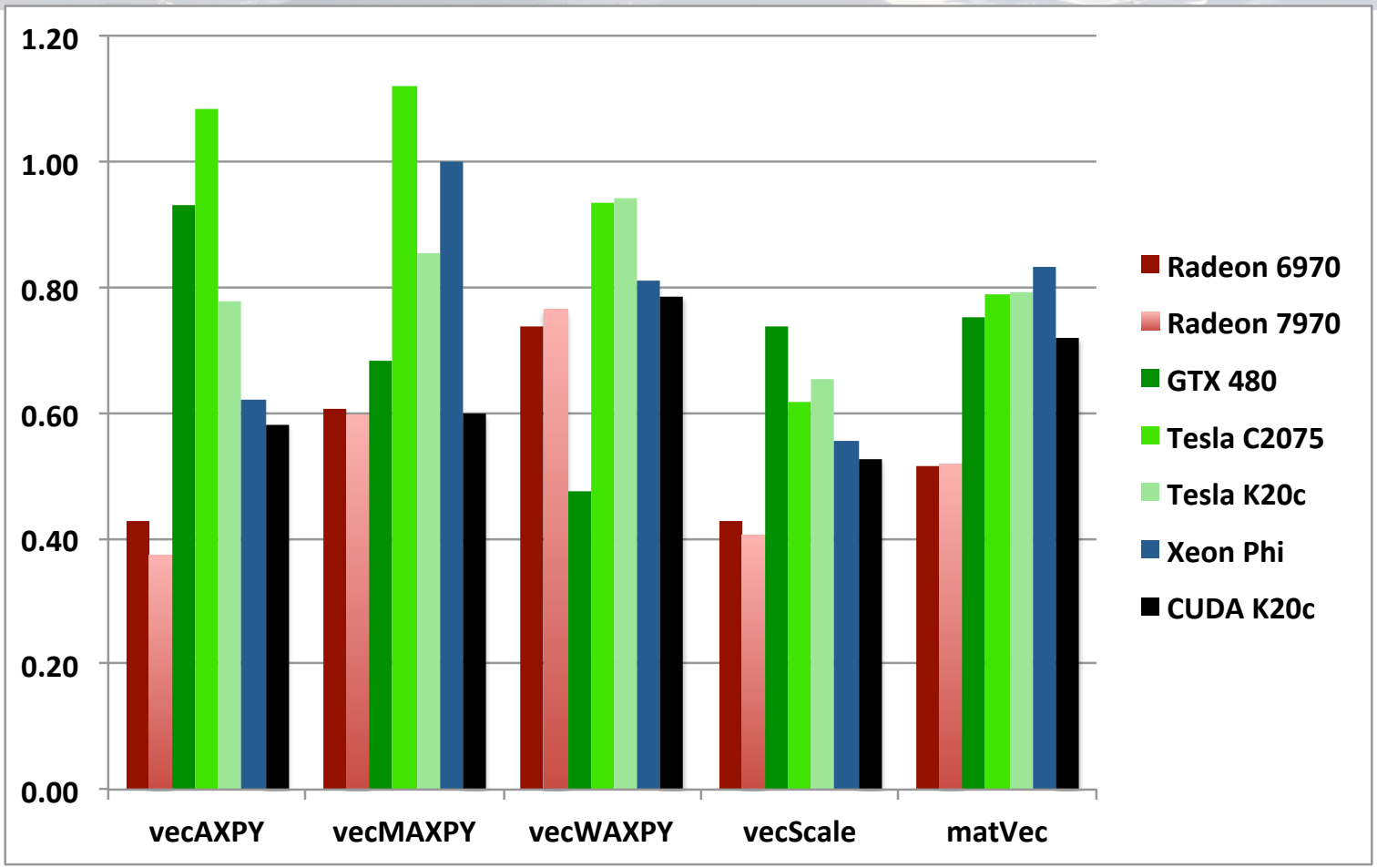
- PFLOTRAN: code for simulation of multiscale, multiphase, multicomponent flow and reactive transport in porous media

Example: Autotuning of PETSc library code (for NVIDIA GPUs)

Example: Structured-grid PDE application using PETSc for GPUs (solid fuel ignition problem). Comparison between library-based (MKL on CPU and Cusp on GPU) implementation and the Orio-tuned implementation (GPU).

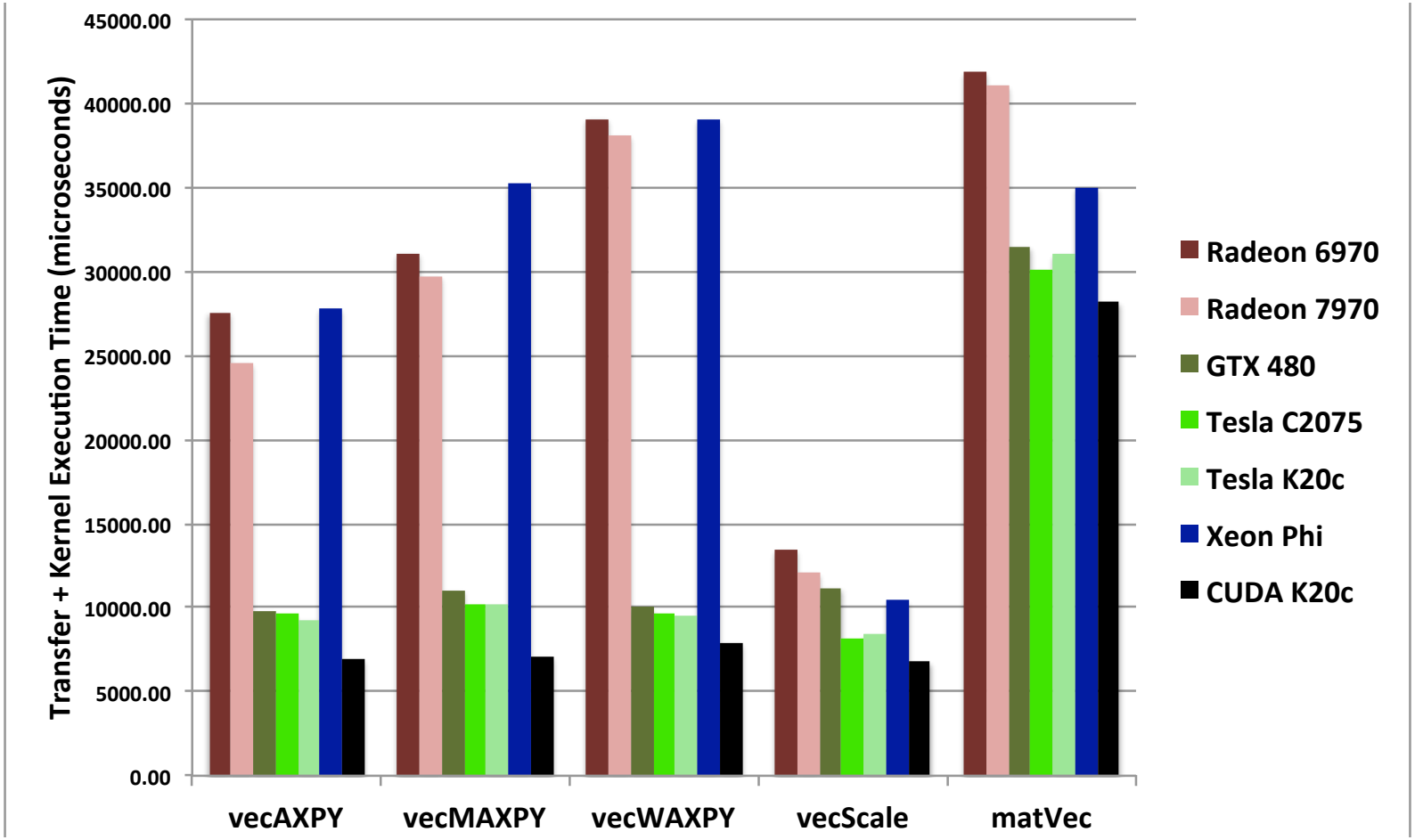


Multi-platform tuning with OpenCL



Performance normalized by time to run the corresponding ViennaCL (http://viennacl.sourceforge.net/) version on the same device. The first six bars in each group correspond to code produced with OrCL, while the last bar is code produced with OrCUDA.

Multi-platform tuning with OpenCL (cont.)



Absolute performance of the **best performing** linear algebra kernel variant across devices. The first six bars in each group correspond to code produced with OrCL, while the last bar is code produced with OrCUDA.

Ongoing and future work

- PBound:
  - Improved cache model
  - Integration with the Eclipse IDE
  - Shared memory parallel codes
  - MPI
- Orio:
  - Vectorization
  - Performance model generation
  - Integration of performance models into search
  - Integration with new BTO tensor contraction language

Conclusion

- Useful performance information can be generated automatically through source code analysis
  - PBound: http://tinyurl.com/PBound
- Both high productivity and high performance can be supported through annotation-based empirical tuning
  - Orio: http://brnorris03.github.io/Orio/
- Thank you!

Extras

UNIVERSITY OF OREGON

CPU optimizations: Stencil computations and dense linear algebra

- Finite-Difference Time-Domain (FDTD) methods
- Composed dense linear algebra operations
  - ATAX

    $$y \leftarrow A^T (A\, x)$$

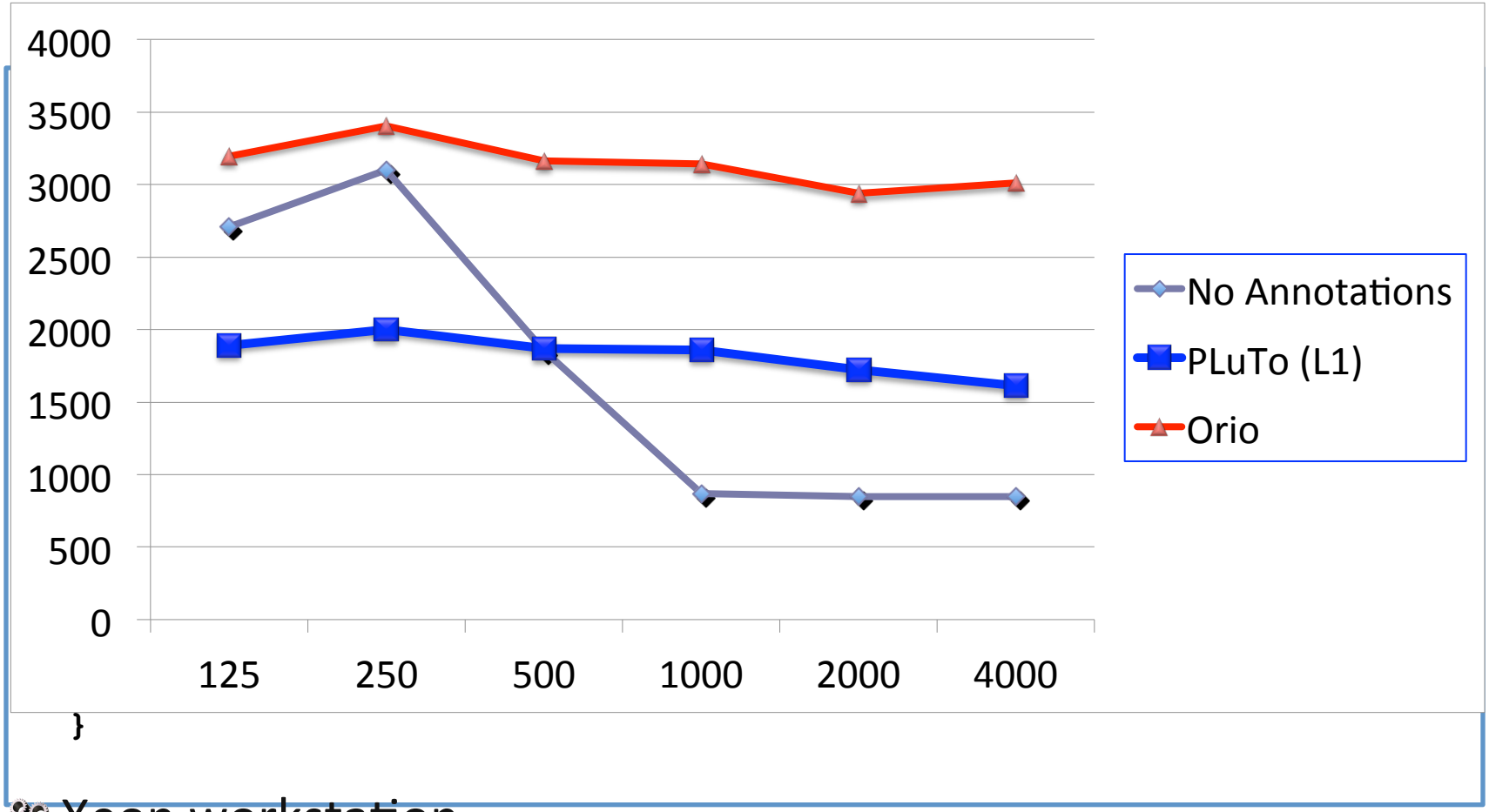
  - GEMVER

    $$A \leftarrow A + u_1\, v_1 + u_2\, v_2$$

    $$x \leftarrow \beta\, A^T\, y + z$$

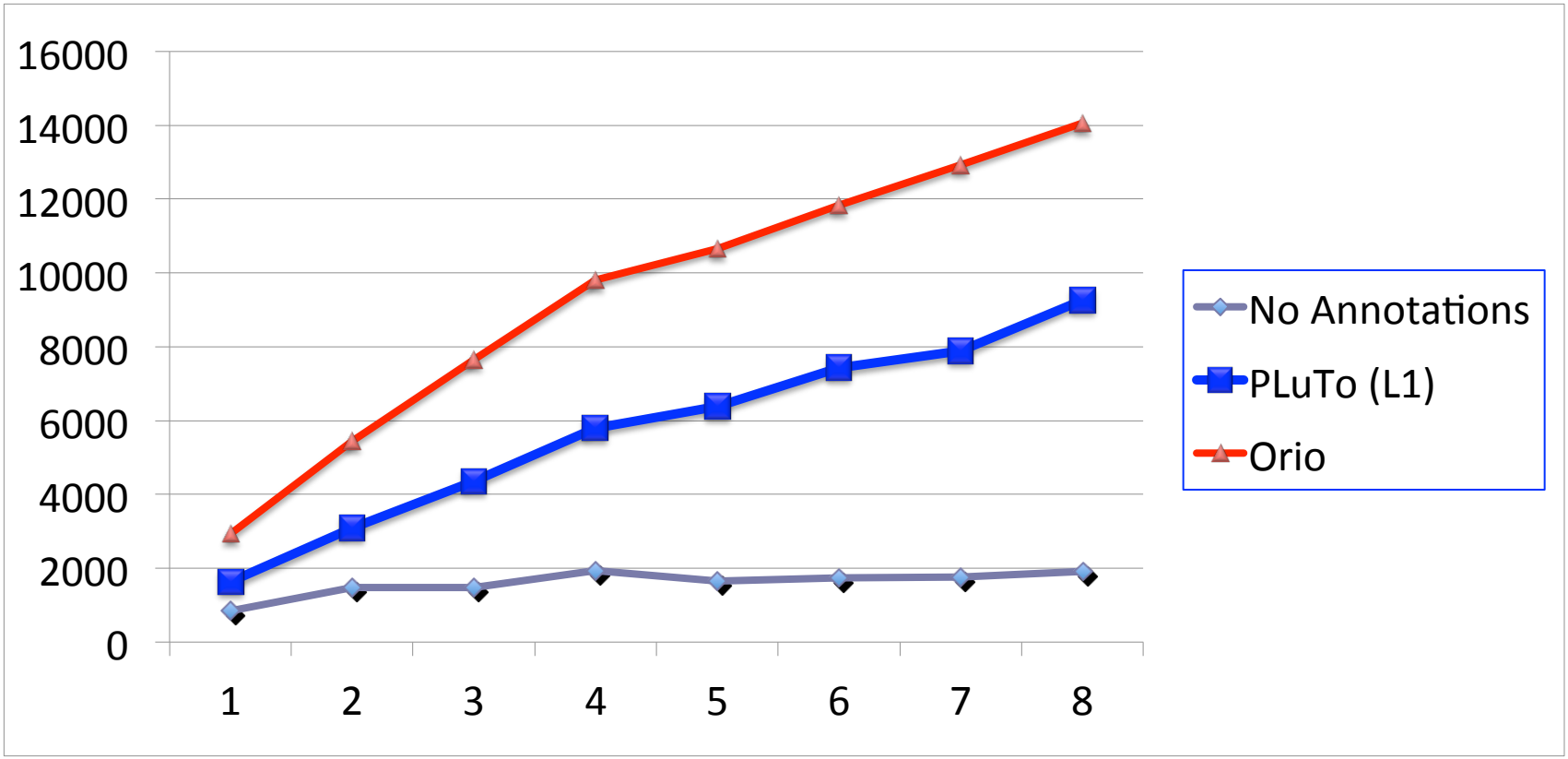
    $$w \leftarrow \alpha\, A\, x$$

2D FDTD: MFLOPs/sec. vs. Array Size (1 Core, tmax=500)



- Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

2D FDTD: MFLOPs/sec. vs. Number of Cores (tmax=500, N=2000)

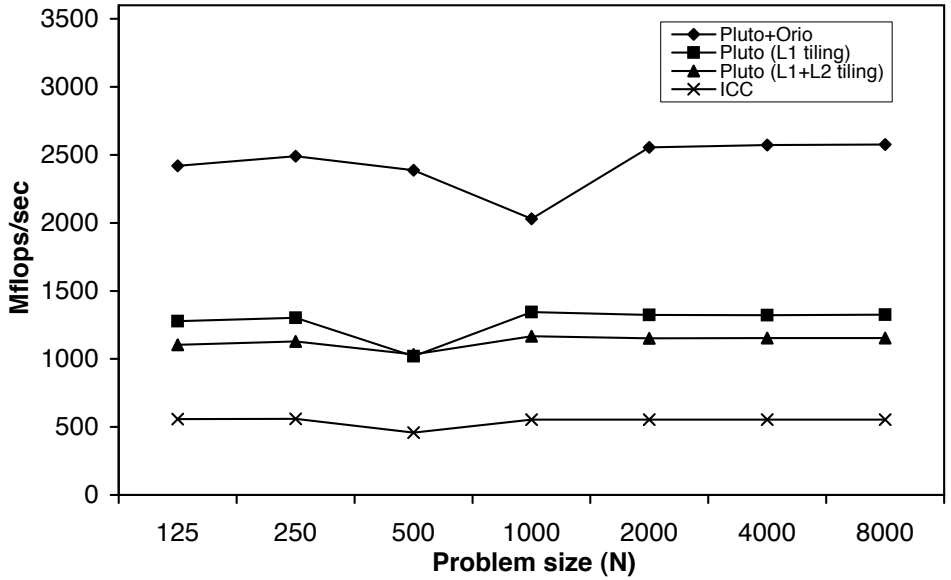


- Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

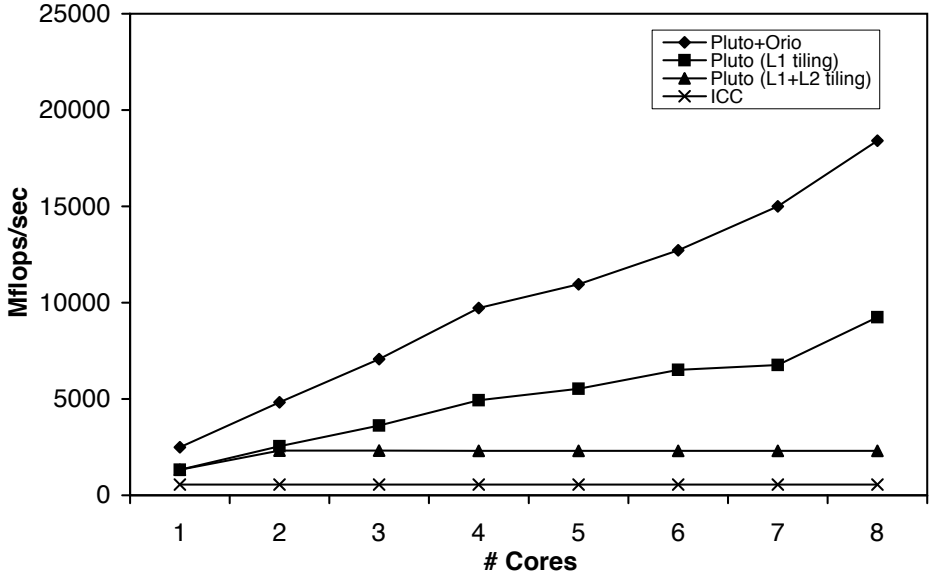
UNIVERSITY OF OREGON

3-D Gauss-Seidel SOR

```
for (t=0; t<=T-1; t++)
  for (i=1; i<=N-2; i++)
    for (j=1; j<=N-2; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1] + A[i][j-1] + A[i][j]
          + A[i][j+1] + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1]) / 9.0;
```



Sequential (T=500)

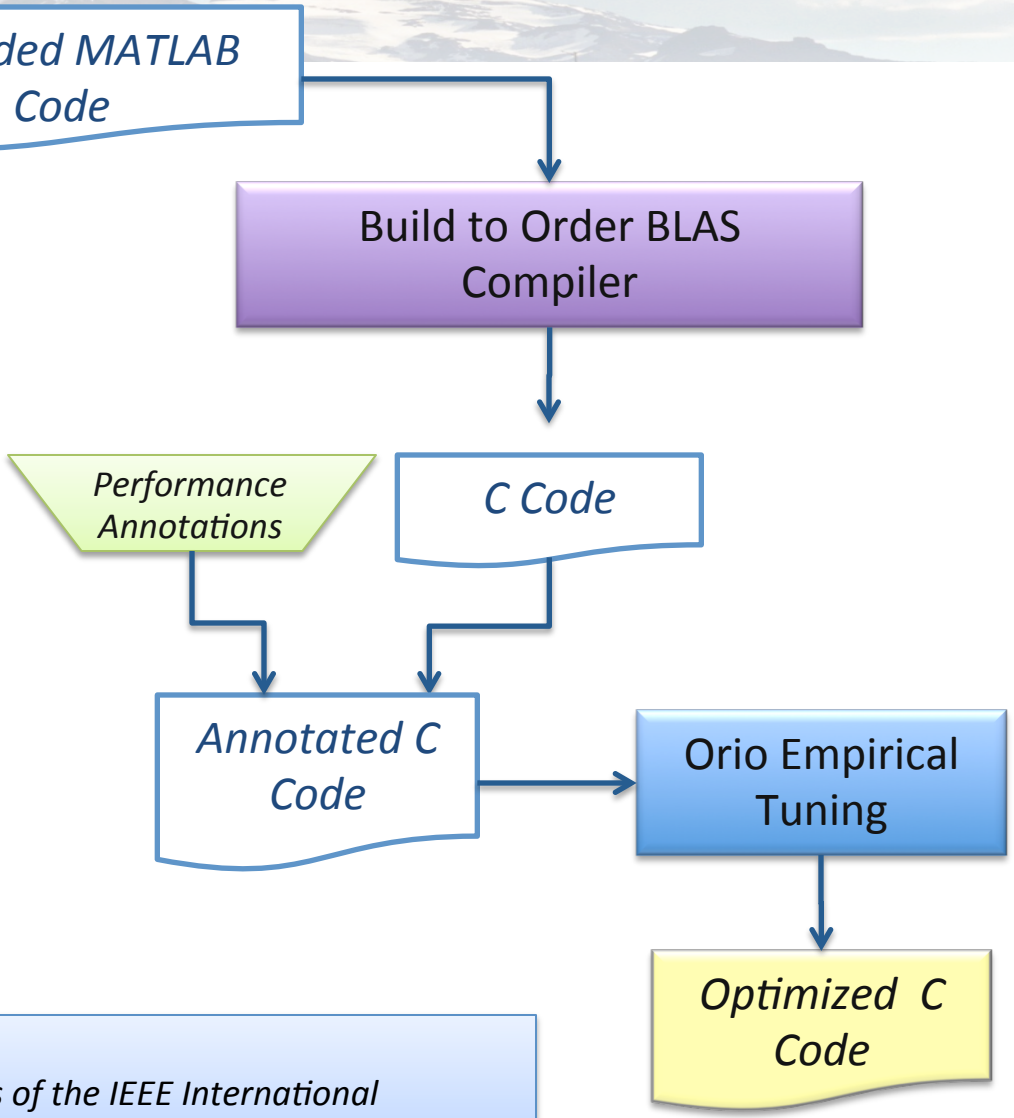


Parallel (T=500, N=4000)

UNIVERSITY OF OREGON

Composed dense linear algebra computations

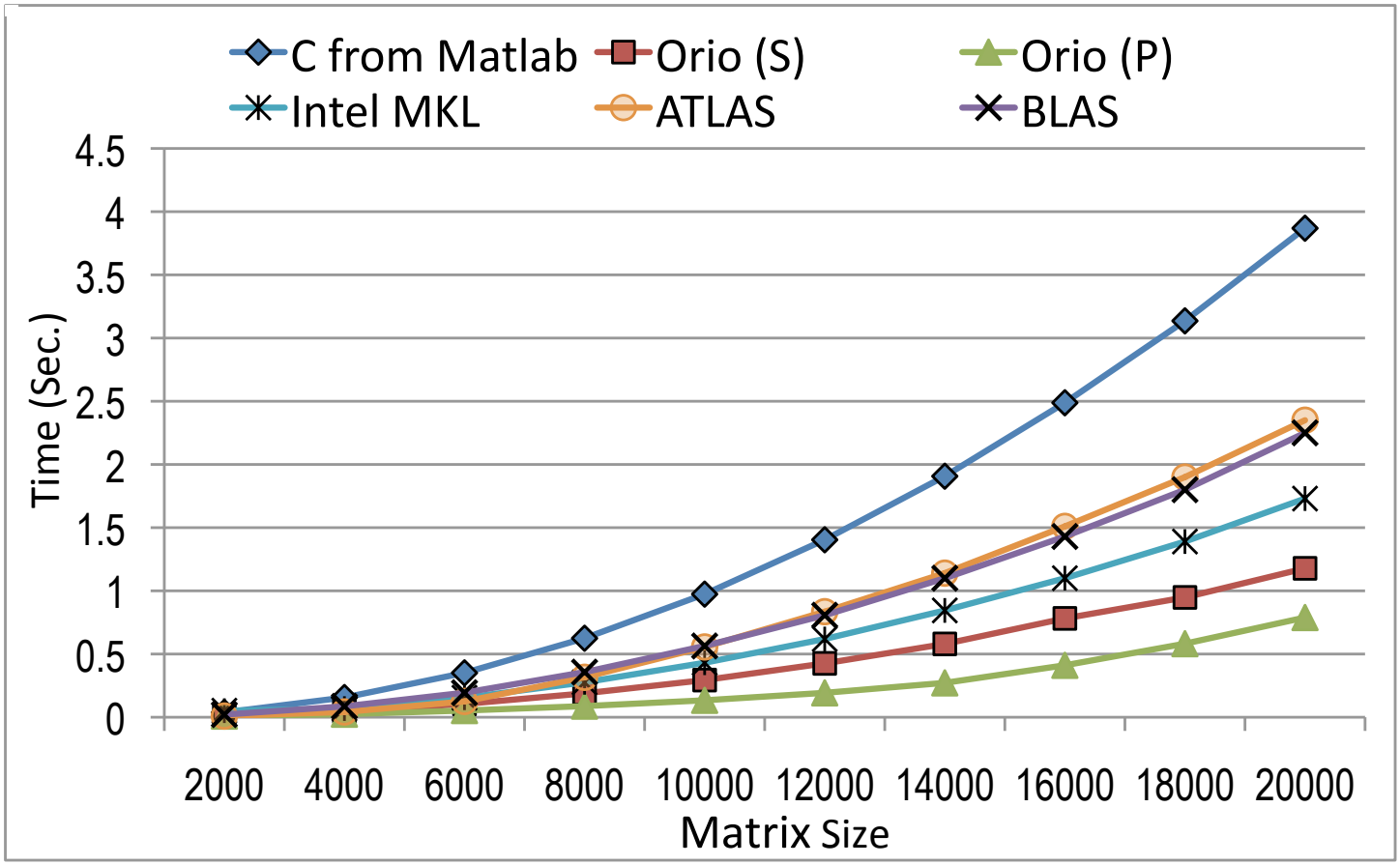
- Productivity:
  - Leverage existing MATLAB compiler technology[1]
- Performance:
  - Annotation language uses slightly extended MATLAB syntax
  - Use existing MATLAB compiler to generate C code
  - Perform analysis of the C code
  - Tune with Orio



[1]Jessup, E., Karlin, I., , Siek, J.:
Build to order linear algebra kernels. *In: Proceedings of the IEEE International Symposium on Parallel and Distributed (IPDPS 2008)*, IEEE (2008) 1–8.

Composed linear algebra: ATAX

```
ATAX
in
  A : row matrix,
  x : vector
out
  y : vector
{
  y = A' * (A * x)
}
```

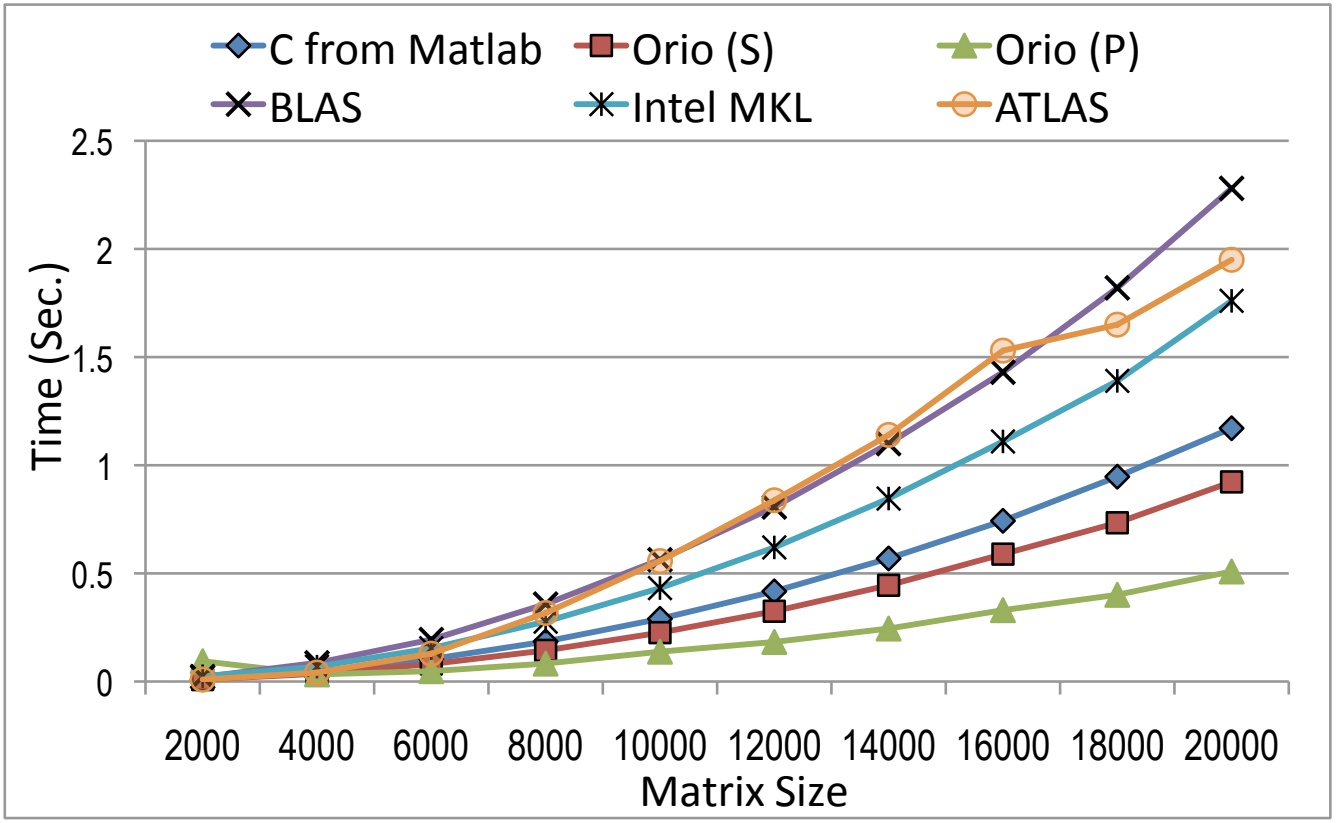


- Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

UNIVERSITY OF OREGON

Composed linear algebra: BiCG Kernel

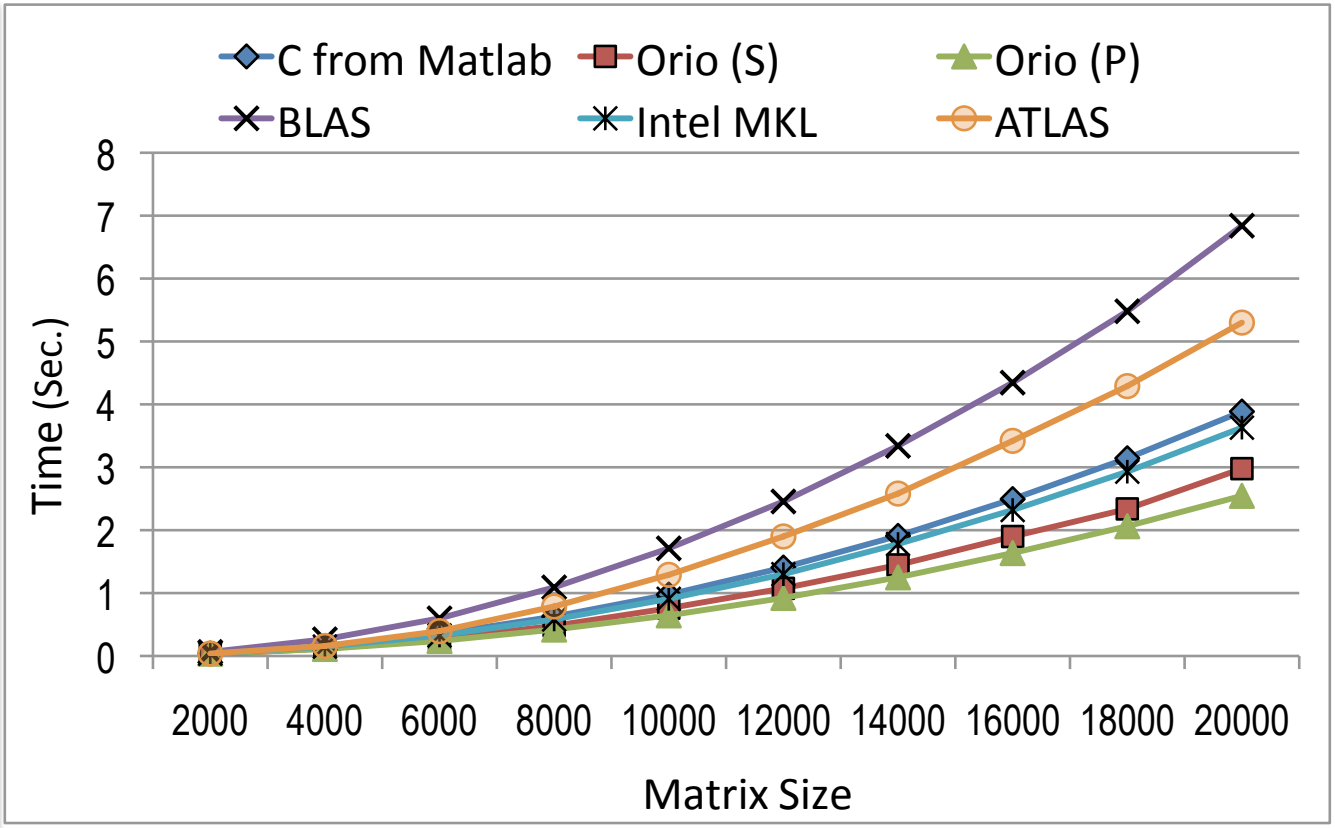
```
BiCG
in
  A : row matrix,
  p : vector, r : vector
out
  q : vector, s : vector
{
  q = A * p
  s = A' * r
}
```



- Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

Composed Linear Algebra: GEMVER

```
GEMVER
in
  A : column matrix,
  u1 : vector, u2 : vector,
  v1 : vector, v2 : vector,
  a : scalar, b : scalar,
  y : vector, z : vector
out
  B : column matrix,
  x : vector, w : vector
{
  B = A + u1 * v1'
          + u2 * v2'
  x = b * (B' * y) + z
  w = a * (B * x)
}
```

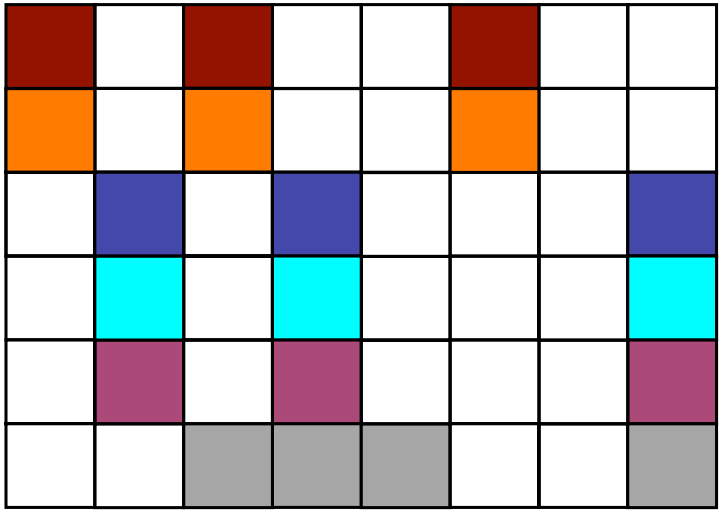


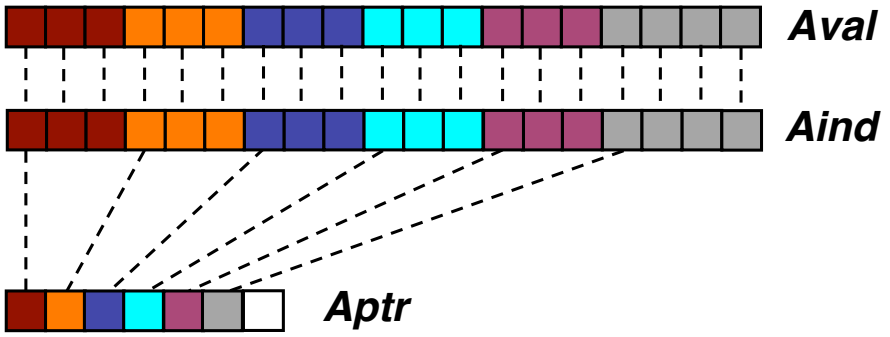
- Xeon workstation (dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64))

Sparse Linear Algebra

- PETSc (http://www.mcs.anl.gov/petsc): a toolkit for the parallel numerical solution of partial differential equations
- Sparse matrix-vector multiplication dominates the performance of many applications

$$\forall A_{i,j} \neq 0 : y_i \leftarrow y_i + A_{i,j} \cdot x_j$$

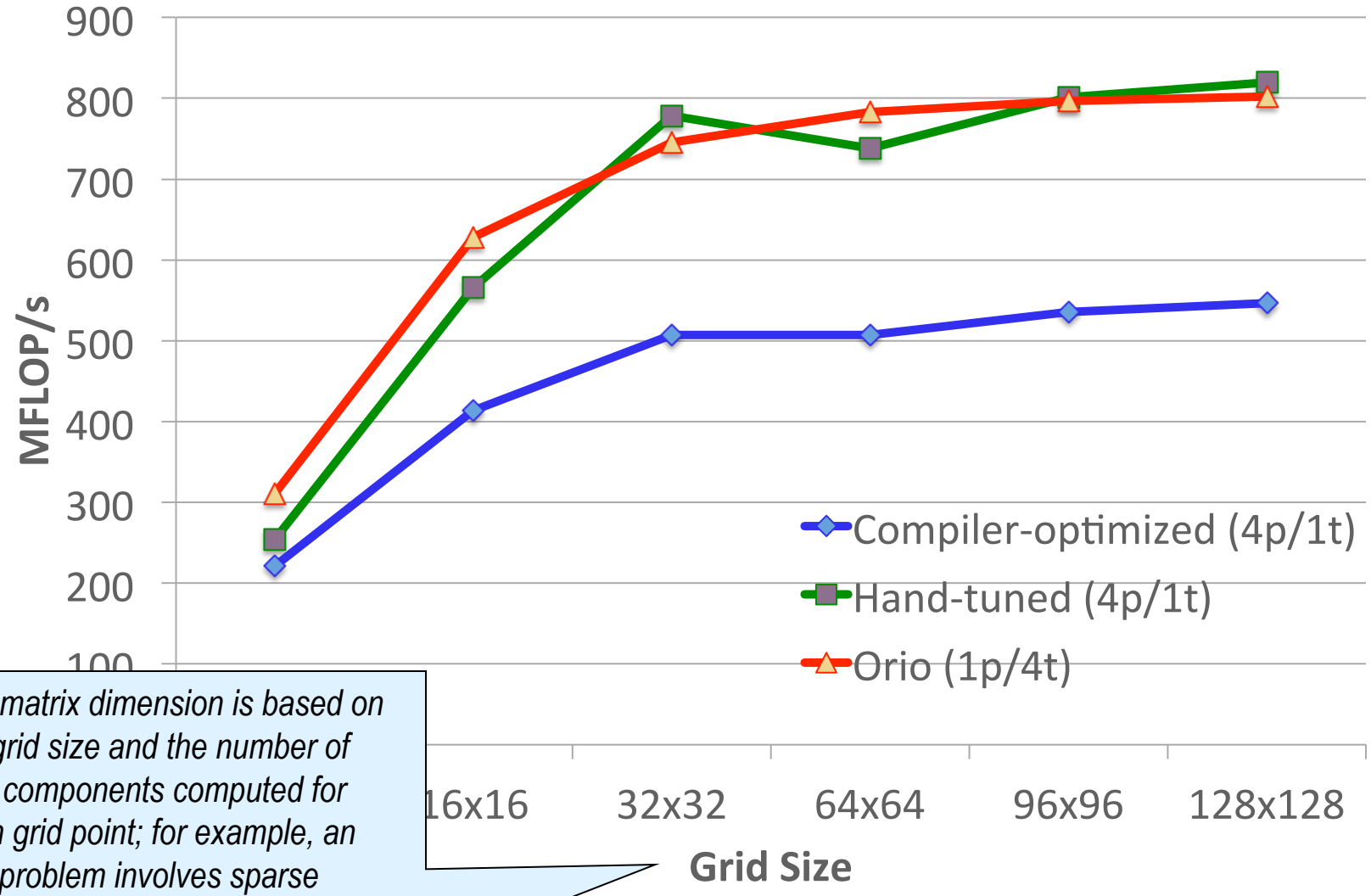





```
for (i=0; i<num_rows; i++)
  for (j=Aptr[i]; j<Aptr[i+1]; j++)
    y[i] += Aval[j]*x[Aind[j]];
```

SpMV in 2D Driven Cavity Flow (8 Nodes)

Blue Gene/P



The matrix dimension is based on the grid size and the number of field components computed for each grid point; for example, an 8×8 problem involves sparse matrices of dimension 900 with 17,040 nonzero entries.

SpMV in a 2D driven cavity flow

