Automating Application Mapping with Autotuning:

Paving the Way to Exascale

Mary Hall

Salishan

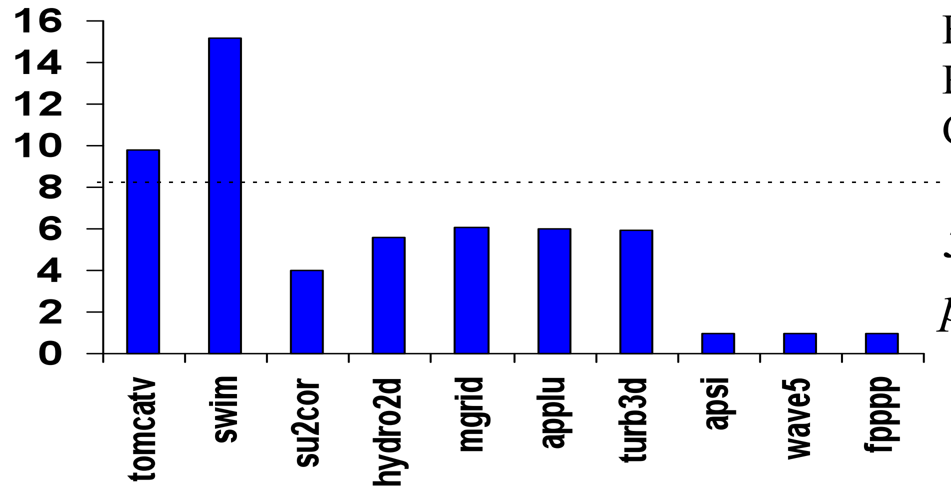
April 2012

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.

Three Goals for Talk

1. Setting expectations for exascale compiler mapping technology from a 20 year retrospective
2. Key issues in future programming models and opportunities for leverage
3. A look at autotuning, a specific enabling technology for exascale

Previous Work in Automatic Parallelization



8-processor Speedups--Digital AlphaServer 8400

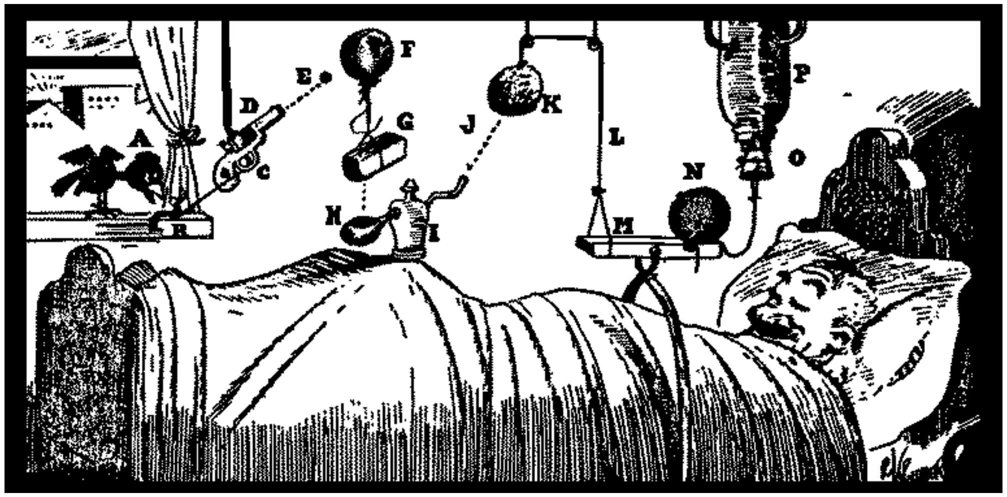
From Hall et al, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, Dec. 1996.

50% higher Specfp95 ratio than previously reported

- Old approaches to compilers mapping parallelism
  - Limited to loops and array computations
  - Difficult to find sufficient granularity (parallel work between synchronization)
  - Very restricted mapping strategy
  - Success but from fragile, complex software

1990s View

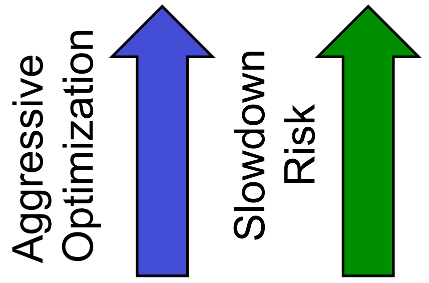
- Programmer writes code at high level
  - Much or all complexity managed by compiler



- But doing everything in the compiler is hard!
  - Expert programmers have knowledge that should be exploited.
  - Compiler development cycle is slow.
  - Application scientists will find expedient solutions.

Historical Organization of Compilers, Users' Perspective

- What's not working
  - Transformations and optimizations often applied in isolation, but significant interactions
  - Static compilers must anticipate all possible execution environments
  - Potential to slow code down; many users say "never use O3"
  - Users write low-level code to get around compiler which makes things even worse



Bottom line: *Known compiler techniques capable of much better performance than they are delivering, but solutions don't generalize across applications and complexity of system is difficult to maintain.*

Compiler Technology: Opportunities and Challenges

Successes:
Vectorization, ILP, scalar optimization, managing registers, code generation, locality optimization (partial)

Still open issues:
Coarse-grain parallelism, high performance for irregular computations, rapid deployment

Opportunities:
Machine resources available for tuning, exascale programming challenges demand fundamental change

Needed advances:
Programming model abstractions partnered with compiler, interface to run-time and programmer

THE UNIVERSITY OF UTAH

Exascale Challenges Will Force Change

- Exascale architectures will be fundamentally different
  - Power management THE fundamental issue
  - Reliability (h/w and s/w) increasingly a concern
  - Memory reduction to .01 bytes/flop
  - Hierarchical, heterogeneous
- Basic rethinking of the software "stack"
  - Express and manage locality and parallelism for ~billion threads
  - Create/support applications that are forward scalable and portable (underlying tools map to h/w details)
  - Manage power and resilience requirements
    - Locality is a big part of power/energy
    - Resilience should leverage abstraction changes

"Software Challenges in Extreme Scale Systems," V. Sarkar, B. Harrod and A. Snavely, SciDAC 2009, June, 2009. Summary of results from a DARPA study entitled, "Exascale Software Study," June 2008 through Feb. 2009.

A View in 2012

Thanks to exascale reports and workshops

- Multiresolution programming systems for different users
  - Joe/Stephanie/Doug **[Pingali, UT]**
  - Elvis/Mort/Einstein **[Intel]**
- Specialization simplifies and improves efficiency
  - Target specific user needs with domain-specific languages/libraries
  - Customize libraries for application needs and execution context
- Interface to programmers and runtime/hardware
  - Seamless integration of compiler with programmer guidance and dynamic feedback from runtime
- Toolkits rather than monolithic systems
  - Layers support different user capability
  - Collaborative ecosystem
- Virtualization (over-decomposition)
  - Hierarchical, or flat but construct hierarchy when applicable?

What is Autotuning?

- Definition:
  - Automatically generate a “search space” of possible implementations of a computation
    - A ***code variant*** represents a unique implementation of a computation, among many
    - A ***parameter*** represents a discrete set of values that govern code generation or execution of a variant
  - Measure execution time and compare
  - Select the best-performing implementation (for exascale, tradeoff between performance/energy/reliability)
- Key Issues:
  - Identifying the search space
  - Pruning the search space to manage costs
  - Off-line vs. on-line search

Three Types of Autotuning Systems

a. Autotuning libraries
   - Library that encapsulates knowledge of its performance under different execution environments
   - Dense linear algebra: **ATLAS, PhiPAC**
   - Sparse linear algebra: **OSKI**
   - Signal processing: **SPIRAL, FFTW**

Current/ Future Work {

b. Application-specific autotuning
   - **Active Harmony** provides parallel rank order search for tunable parameters and variants
   - **Sequoia** and **PetaBricks** provide language mechanism for expressing tunable parameters and variants

c. Compiler-based autotuning (this talk!)
   - Other examples: Saday et al., Swany et al., Eignenmann et al.
   - Related concepts: iterative compilation, continuous compilation, learning-based compilation

THE UNIVERSITY OF UTAH

Differences: Present and Future

Who/What	Present	Future
Application programmer writes	A single implementation of a computation, or perhaps a few guarded by run-time tests	A compact search space of parameterized variants
Library developer writes	Numerous implementations of a computation, guarded by run-time tests	A compact search space of parameterized variants
Compiler generates	A single implementation of a computation, or perhaps a few guarded by run-time tests	A compact search space of parameterized variants
System executes	Compiled code as provided	A synthesis of variants and their parameter values meeting optimization criteria

Compiler-Based Autotuning: My Philosophy

- ***Foundational Concepts***
  - Identify search space through a high-level description that captures a large space of possible implementations
  - Prune space through compiler domain knowledge and architecture features
  - Provide access to programmers with **transformation recipes** (controversial)
  - Uses source-to-source transformation for portability, and to leverage vendor code generation
  - Requires ***restructuring of the compiler***
- ***Impact***
  - Developers write less and higher-level code, more automatically generated/managed
  - Systematic characterization and analysis

THE UNIVERSITY OF UTAH

Transformation Recipes for Autotuning: Incorporate the Best Ideas from Manual Tuning

Nvidia GTX-280 implementation
Mostly corresponds to CUBLAS 2.x and Volkov's SC08 paper

```
tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
              {"ii","jj","i","j"})
tile_by_index({"k"},{TK},{l1_control="kk"},
              {"ii","jj","kk","i","j","k"},strided)
tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="t"},
              {"ii","jj","kk","t","tt","j","k"})
cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
        {block={"ii","jj"},thread={"t","tt"}})
copy_to_shared("tx","b",-16)
copy_to_registers("kk","c")
copy_to_texture("b")
unroll_to_depth(2)
```

Nvidia TC2050 Fermi implementation
Mostly corresponds to CUBLAS 3.2 and MAGMA

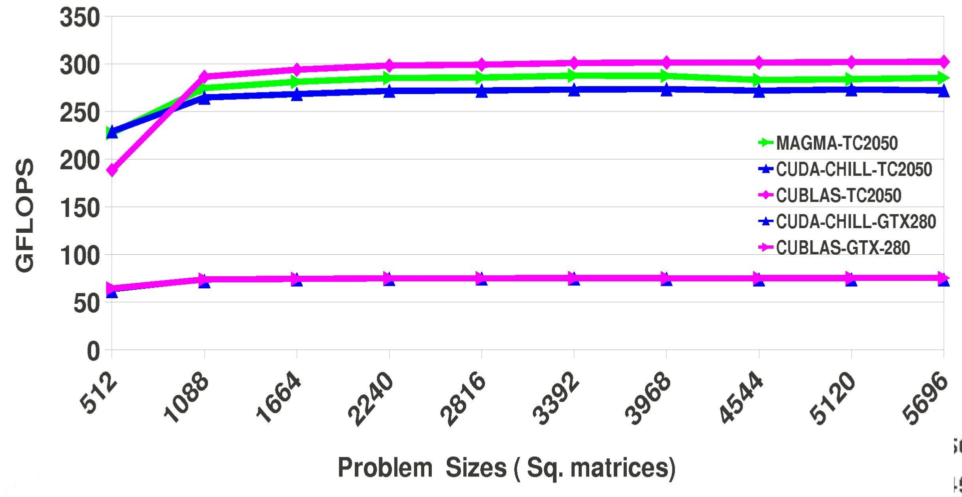
```
tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
              {"ii","jj","i","j"})
tile_by_index({"k"},{TK},{l1_control="kk"},
              {"ii","jj","kk","i","j","k"},strided)
tile_by_index({"i"},{TK},{l1_control="t",l1_tile="tt"},
              {"ii","jj","kk","tt","t","j","k"})
tile_by_index({"j"},{TK},{l1_control="s",l1_tile="ss"},
              {"ii","jj","kk","tt","t","ss","s","k"})
cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
        {block={"ii","jj"},thread={"tt","ss"}})
copy_to_shared("tx","b",-16)
copy_to_texture("b")
copy_to_shared("tx","a",-16)
copy_to_texture("a")
copy_to_registers("kk","c")
unroll_to_depth(2)
```

Different computation decomposition leads to additional tile command

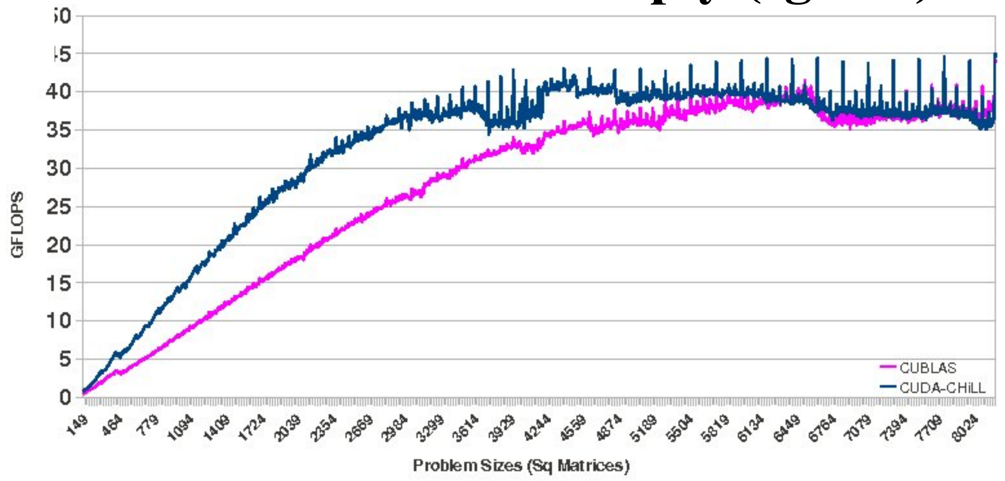
a in shared memory, both a and b are read through texture memory

Compiler + Autotuning can yield comparable and even better performance than manually-tuned libraries

Matrix-Matrix Multiply (dgemm)



Matrix-Vector Multiply (sgemv)

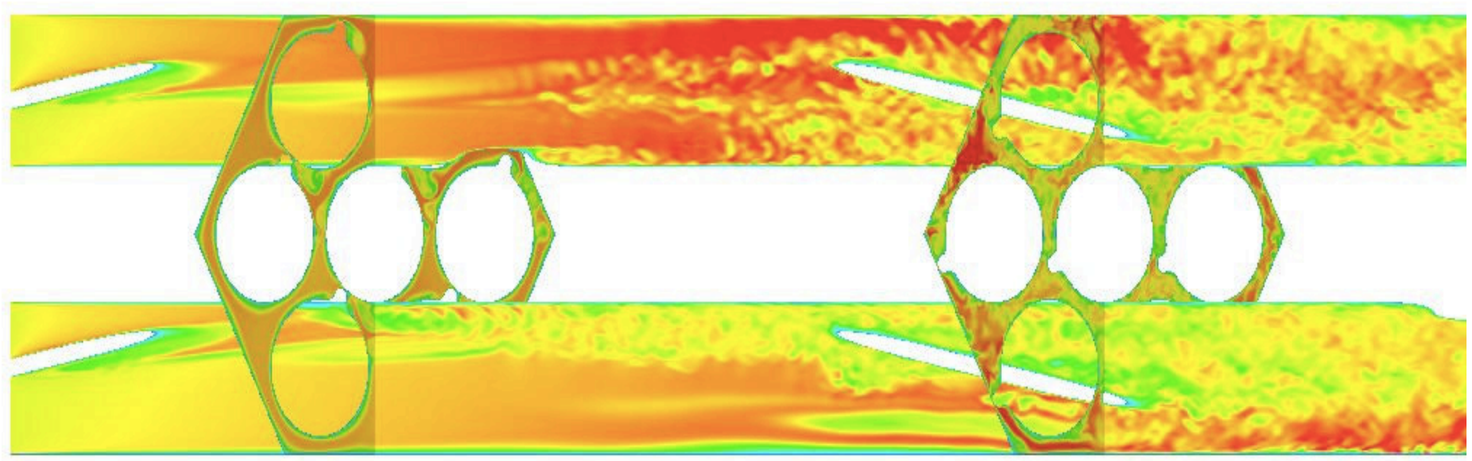


- Performance comparison with CUBLAS 3.2

"Autotuning, Code Generation and Optimizing Compiler Technology For GPUs," M. Khan, PhD Dissertation, University of Southern California, May 2012.

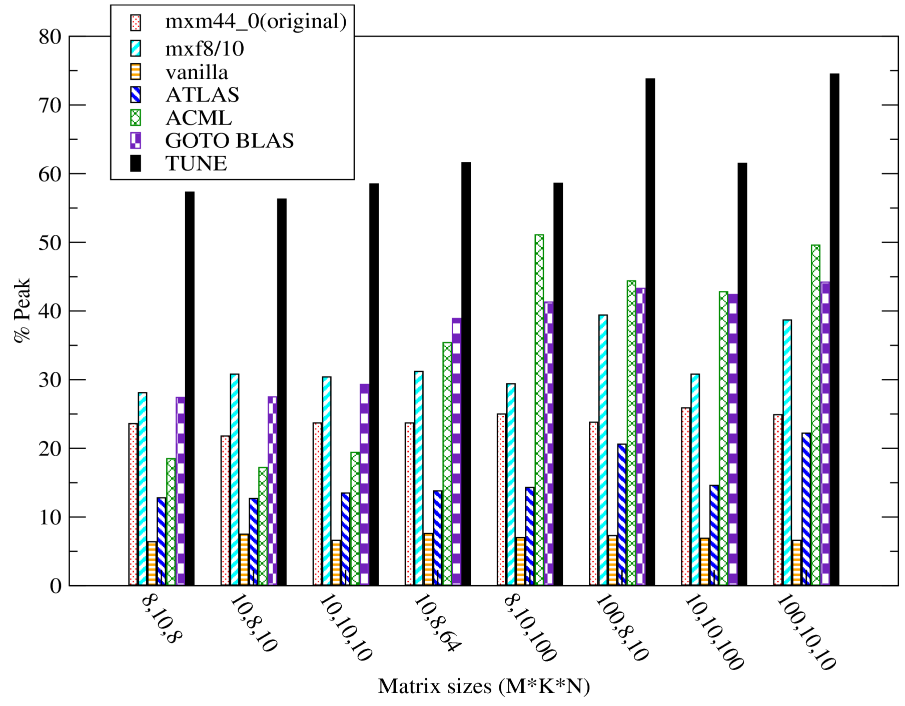
Autotuning and Specialization for Nek5000

Spectral element code: turbulence in wire-wrapped subassemblies



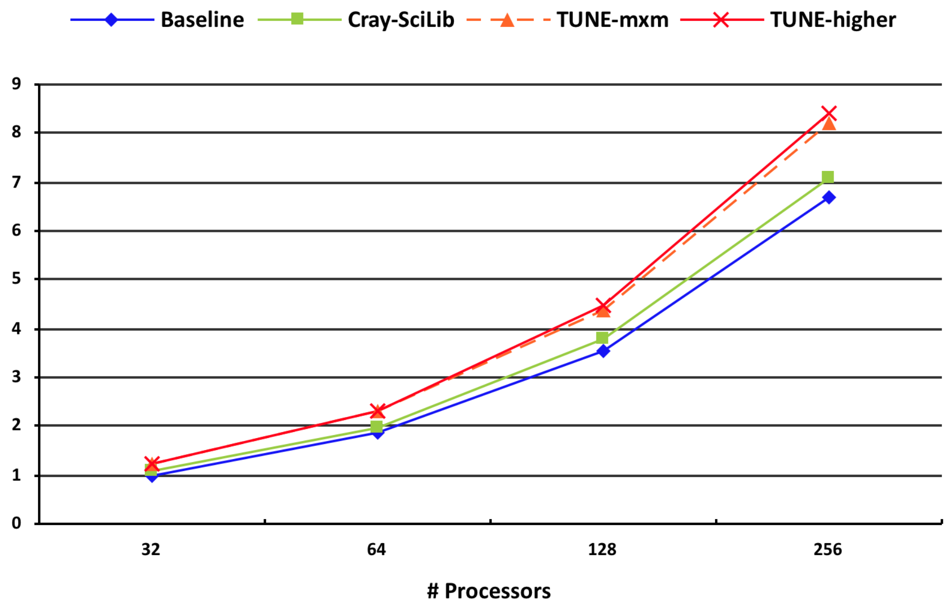
- Applications: nuclear energy, astrophysics, ocean modeling, combustion, bio fluids, ...
- Scales to P > 10,000 (Cray XT5, BG/P)
- > 75% of time spent on manually optimized mxm
  - matrix multiply of very small, rectangular matrices
  - matrix sizes remain the same for different problem sizes

nek5000: Automatically-Generated BLAS Code is Faster than Manually-Tuned Libraries



Library:
2.2X speedup for specialized DGEMM

Application:
26% performance gain on Jaguar



"Autotuning and Specialization: Speeding up Nek5000 with Compiler Technology," J. Shin, M. W. Hall, J. Chame, C. Chen, P. Fischer, P. D. Hovland, International Conference on Supercomputing, June, 2010.

THE UNIVERSITY OF UTAH

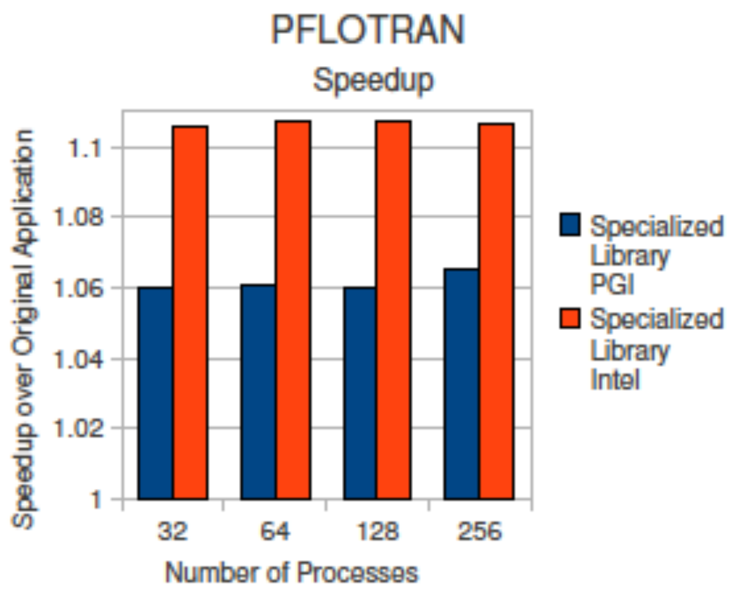
Autotuning and Specialization also Benefit PETSc Sparse Libraries

- PFloTran, an example from PERI
  - Models Multiscale-Multiphase-Multicomponent Subsurface Reactive Flows
- PETSc routines comprise 30% of execution time on hopper at NERSC. Two routines achieve only 4% of peak.

Example: MatSolve_SeqBAIJ_N

Represents sparse matrix as collection of dense blocks

PETSc includes a large number of different implementations specialized for different block sizes



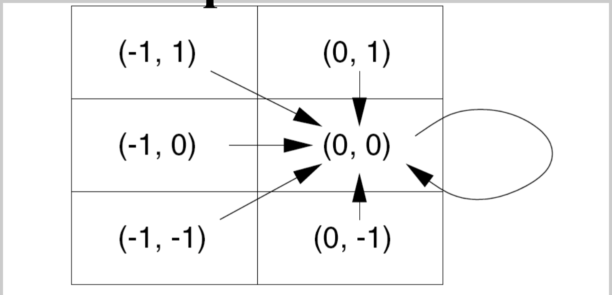
"Improving High-Performance Sparse Libraries using Compiler-Assisted Specialization: A PETSc Case Study", S. Ramalingam, M. W. Hall, C. Chen, Workshop on High-Level Programming Models and Supporting Systems, May, 2012.

Application example from PERI: SMG2000 Optimization

- Semi-coarsening multigrid on structured grids
  - Residual computation contains sparse matrix-vector multiply bottleneck, expressed in 4-deep loop nest
  - Key computation identified by HPCToolkit

```
for si = 0 to NS-1
  for k = 0 to NZ-1
    for j = 0 to NY-1
      for i = 0 to NX-1
        r[i + j*JR + k*KR] -=
              A[i + j*JA + k*KA + SA[si]]
            * x[i + j*JX + k*KX + Sx[si]]
```

2D 6-point Stencil



$S = \{(-1,1),(-1,0),(-1,-1),(0,1),(0,0),(0,-1)\}$

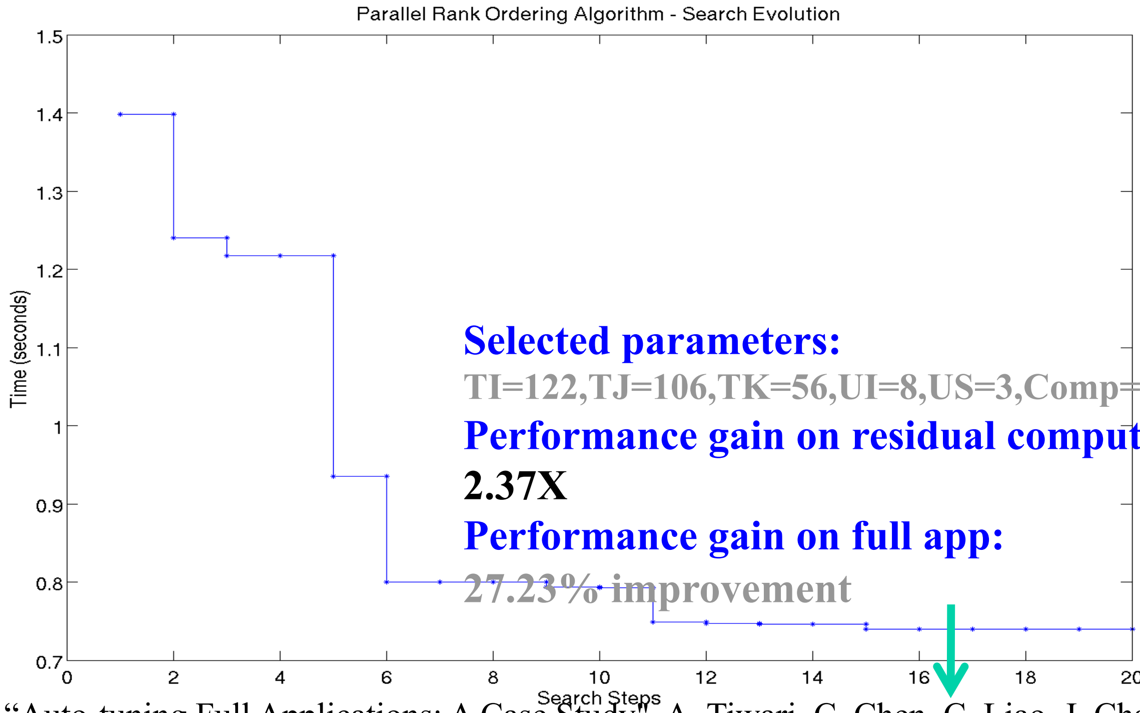
Parallel Heuristic-Based Search for SMG2000 Converges Rapidly

Outlined Code (from ROSE outliner)

```
for (si = 0; si < stencil_size; si++)
   for (kk = 0; kk < hypre__mz; kk++)
      for (jj = 0; jj < hypre__my; jj++)
         for (ii = 0; ii < hypre__mx; ii++)
            rp[((ri+ii)+(jj*hypre__sy3))+(kk*hypre__sz3)] -=
               ((Ap_0[((ii+(jj*hypre__sy1))+ (kk*hypre__sz1))+
                (((A->data_indices)[i])[si])])*
                (xp_0[((ii+(jj*hypre__sy2))+(kk*hypre__sz2))+(( *dxp_s)[si])]));
```

CHiLL Transformation Recipe

```
permute([2,3,1,4])
tile(0,4,TI)
tile(0,3,TJ)
tile(0,3,TK)
unroll(0,6,US)
unroll(0,7,UI)
```



Selected parameters:
TI=122,TJ=106,TK=56,UI=8,US=3,Comp=gcc
Performance gain on residual computation:
2.37X
Performance gain on full app:
27.23% improvement

Optimization search space has 581M points!

Parallel search (Active Harmony) evaluates 490 points, converges in 20 steps

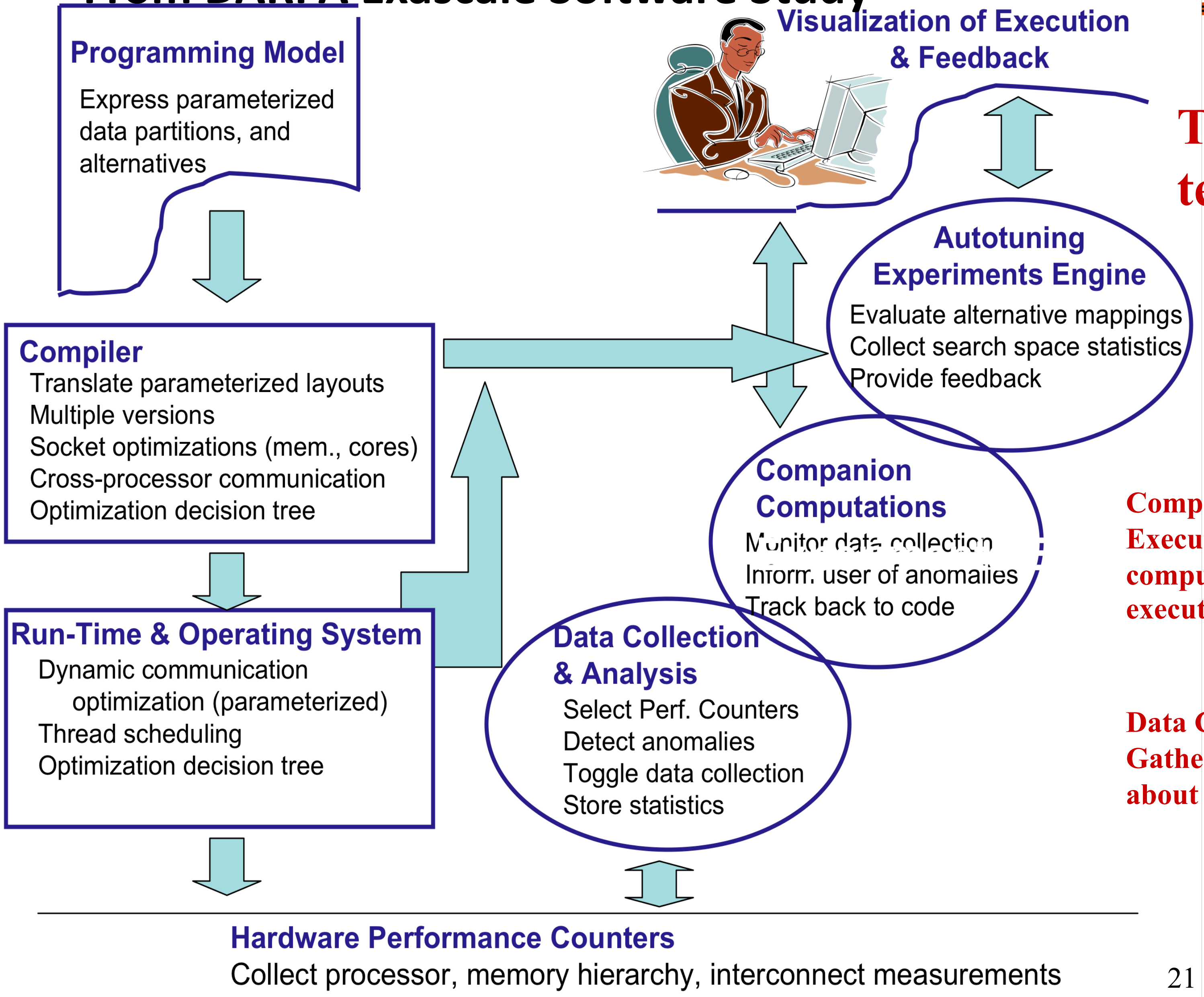
"Auto-tuning Full Applications: A Case Study", A. Tiwari, C. Chen, C. Liao, J. Chame, J. Hollingsworth, M. Hall and D. Quinlan, International Journal of High Performance Computing Applications, 25(3):286-294, Aug. 2011.

Summary: Autotuning Challenges

- **Conceptual:** Rethink the development process as a way of expressing a search space rather than a fixed implementation
  - What are the right abstractions to expose to programmer
  - Integrate into multiresolution system
- Navigating prohibitively large search space
  - Includes performance, power and reliability
  - Models and pruning are critical
  - Parallel search algorithms can be effective
  - Tuning multiple computations simultaneously still an open problem
- Managing overhead (performance, storage and energy)

Example Tools Scenario: Optimizing Data Decomposition

From DARPA Exascale Software Study



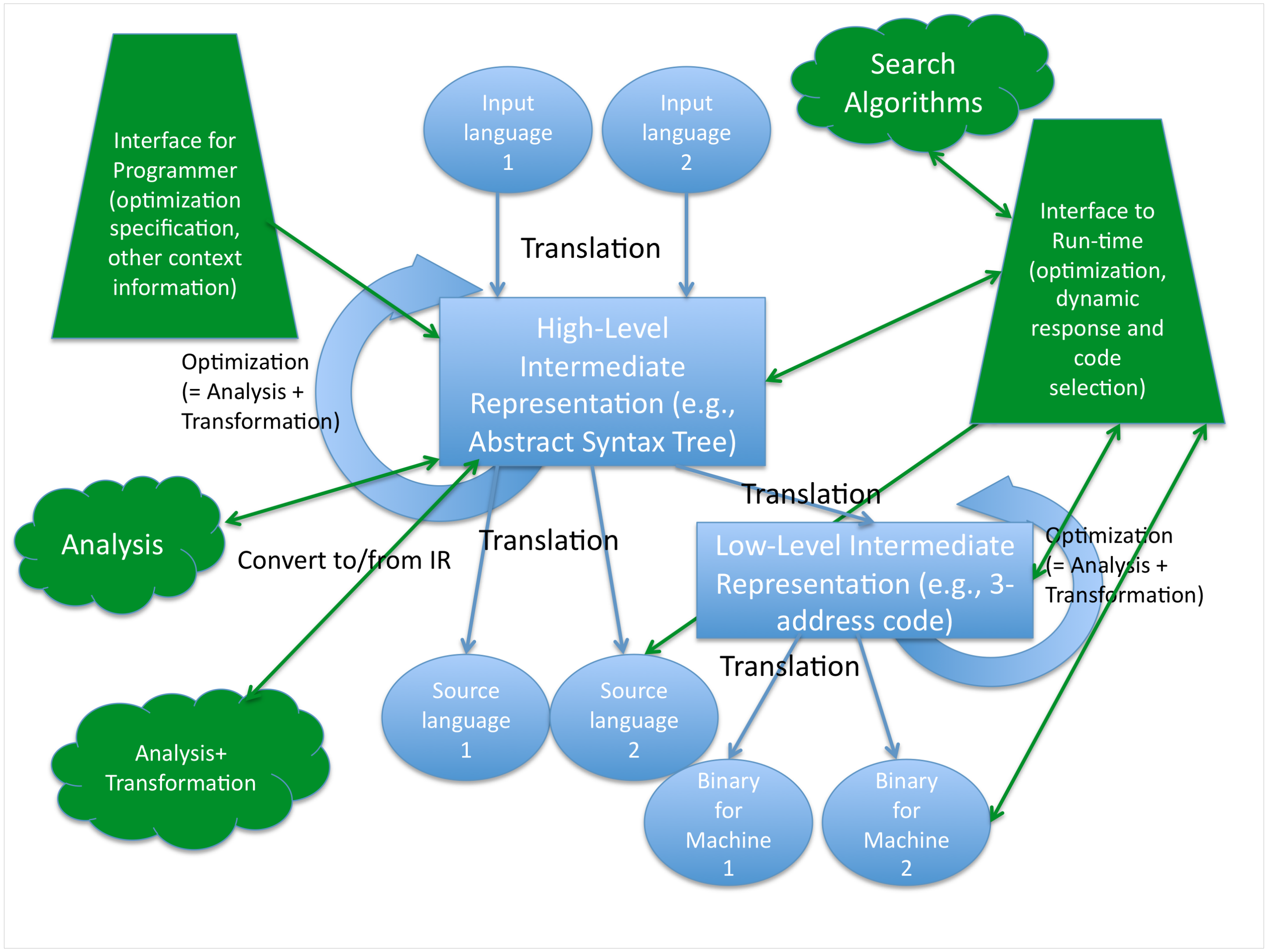
Programming Model
Express parameterized data partitions, and alternatives
Visualization of Execution & Feedback
Compiler
Translate parameterized layouts
Multiple versions
Socket optimizations (mem., cores)
Cross-processor communication
Optimization decision tree
Autotuning Experiments Engine
Evaluate alternative mappings
Collect search space statistics
Provide feedback
Companion Computations
Monitor data collection
Inform user of anomalies
Track back to code
Run-Time & Operating System
Dynamic communication optimization (parameterized)
Thread scheduling
Optimization decision tree
Data Collection & Analysis
Select Perf. Counters
Detect anomalies
Toggle data collection
Store statistics
Hardware Performance Counters
Collect processor, memory hierarchy, interconnect measurements

Three enabling technologies

Autotuning: Automatically explore search space of implementation alternatives

Companion Computations: Execute collaboratively with computation to improve its execution

Data Collection and Analysis: Gather and synthesize data about execution



Interface for Programmer (optimization specification, other context information)
Input language 1
Input language 2
Translation
Search Algorithms
Interface to Run-time (optimization, dynamic response and code selection)
High-Level Intermediate Representation (e.g., Abstract Syntax Tree)
Optimization (= Analysis + Transformation)
Analysis
Convert to/from IR
Translation
Translation
Low-Level Intermediate Representation (e.g., 3-address code)
Optimization (= Analysis + Transformation)
Translation
Analysis+ Transformation
Source language 1
Source language 2
Binary for Machine 1
Binary for Machine 2