Compiler-Based Autotuning for Productive Parallel Programming *and its Relationship to Design Space Exploration*

Mary Hall

May, 2011

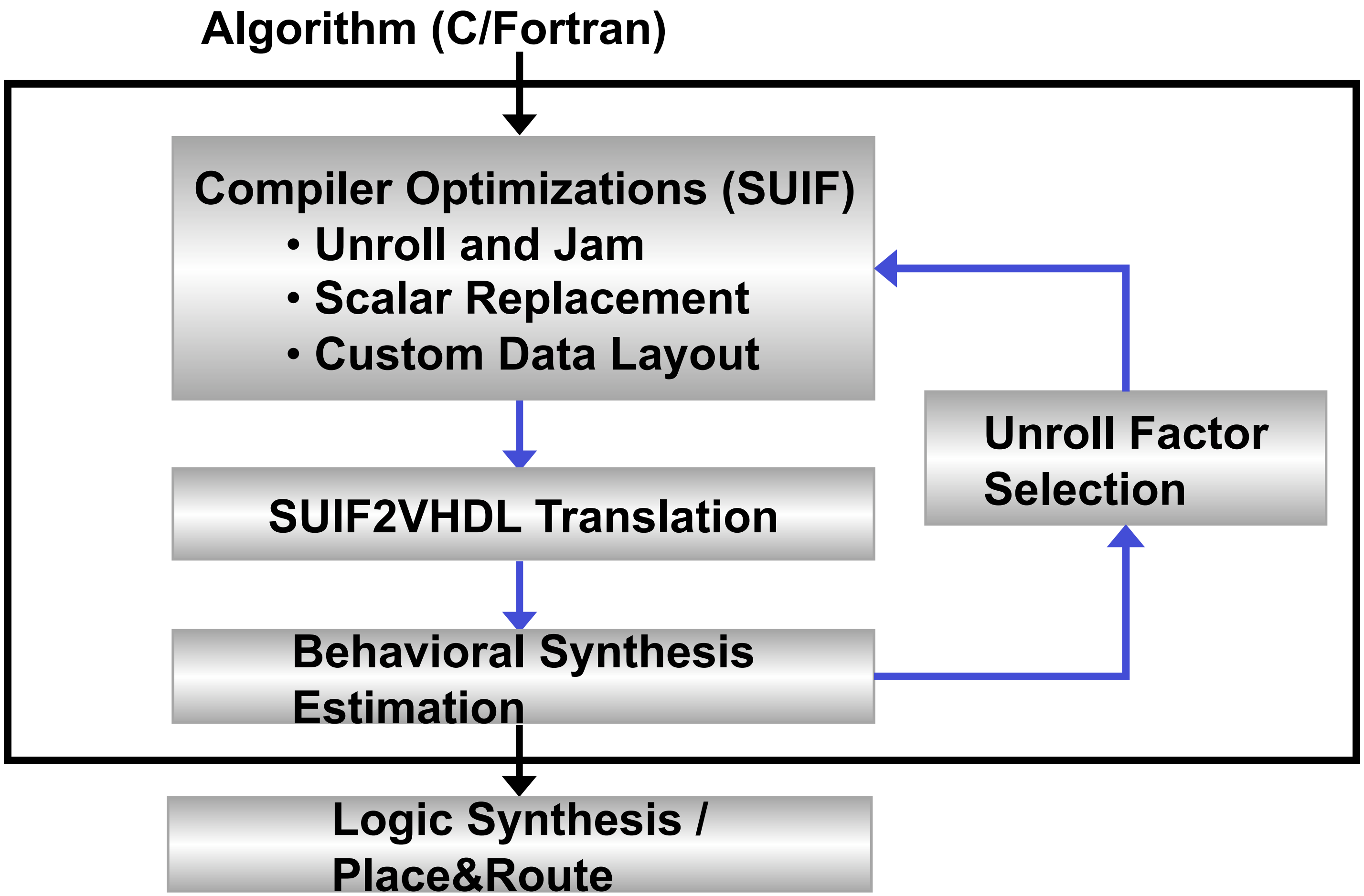
*** This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.**

THE UNIVERSITY OF UTAH

Outline

1. FPGA design space exploration in DEFACTO
2. Related idea: Compiler-based autotuning in CHiLL and CUDA-CHiLL
3. Case study 1: specialized linear algebra for supercomputer application
4. Case study 2: linear algebra for GPUs

Automatic Design Space Exploration in DEFACTO

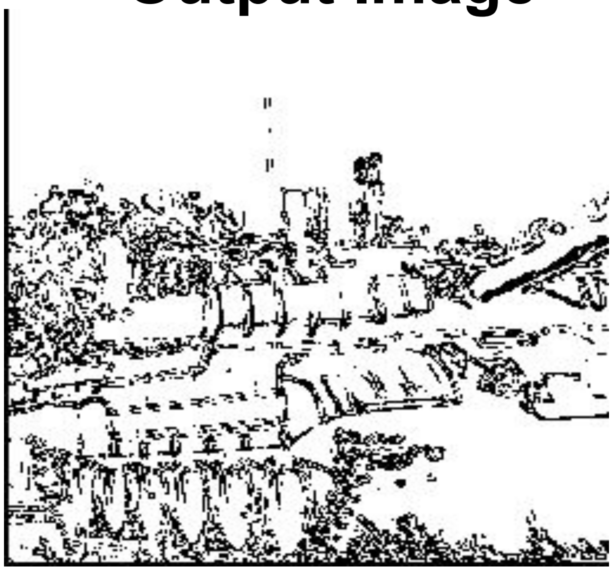


- Overall, less than 2 hours
- 5 minutes for optimized design selection

B. So, M. Hall and P. Diniz, "A Compiler Approach for Fast Design Space Exploration for FPGA-Based Systems," ACM PLDI '02, June, 2002.

Sobel Edge Detection on Annapolis Wildstar

Input Image | **Output Image**



Metrics	**Manual**	**Automated**
Space (slices)	2238	2279 (2% increase)
Cycles	326K	518K
Clock Rate (MHz)	42	40
Execution Time (one frame)	7.7 ms (100%)	12.95 ms (159%)
Design Time	about 1 week	42 minutes

DEFACTO Major Findings

- Performance within 2X of manual, 100X reduction in design time and automatic [M&M05].
- Design space exploration using compiler optimizations of C code, VHDL generation, then synthesis [PLDI02].
  - Unrolling to exploit parallelism, scalar replacement to manage storage, loop permutation
- Estimates from behavioral synthesis, although inaccurate, provide fast and relative guidance on design selection [DAC03].
- Extended to pipelined designs using communication analysis and design space exploration [FCCM02,DAC03,FPGA05].

DEFACTO collaborators: primarily Byoungro So, Heidi Ziegler, Joonseok Park and Pedro Diniz

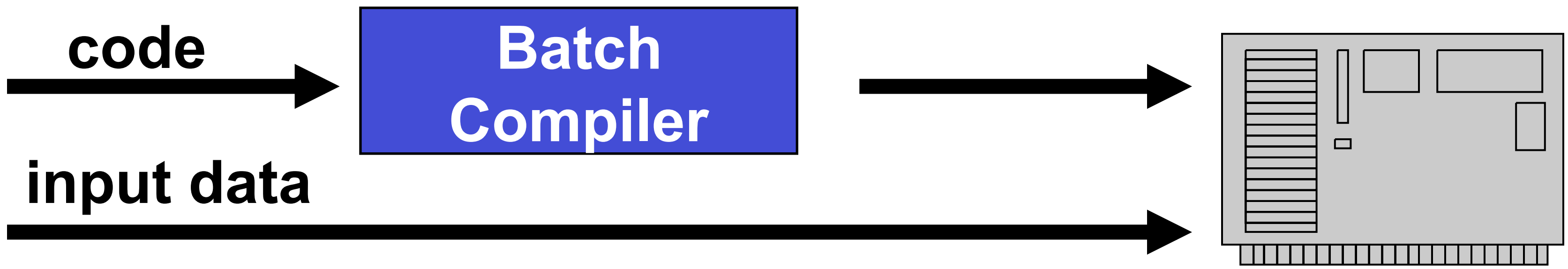
From Custom Hardware to Tuned Software

- Architectures are getting increasingly complex
  - Multiple cores, deep memory hierarchies, software-controlled storage, shared resources, SIMD compute engines, heterogeneity, ...
- Performance optimization is getting more important
  - Today's sequential and parallel applications ***may not*** be faster on tomorrow's architectures.
  - Especially if you want to add new capability!
  - Managing ***data locality*** even more important than parallelism.
  - Managing ***power*** of growing importance, too.

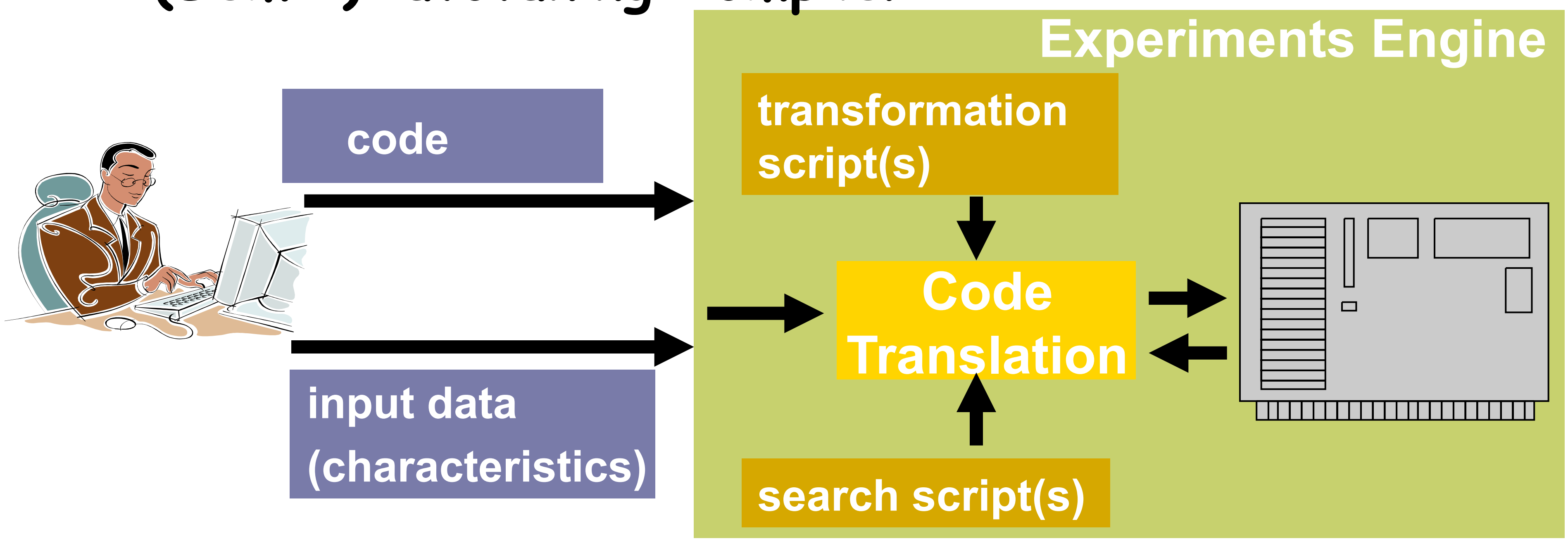
Complexity!

Motivation: Collaborative Autotuning "Compiler"

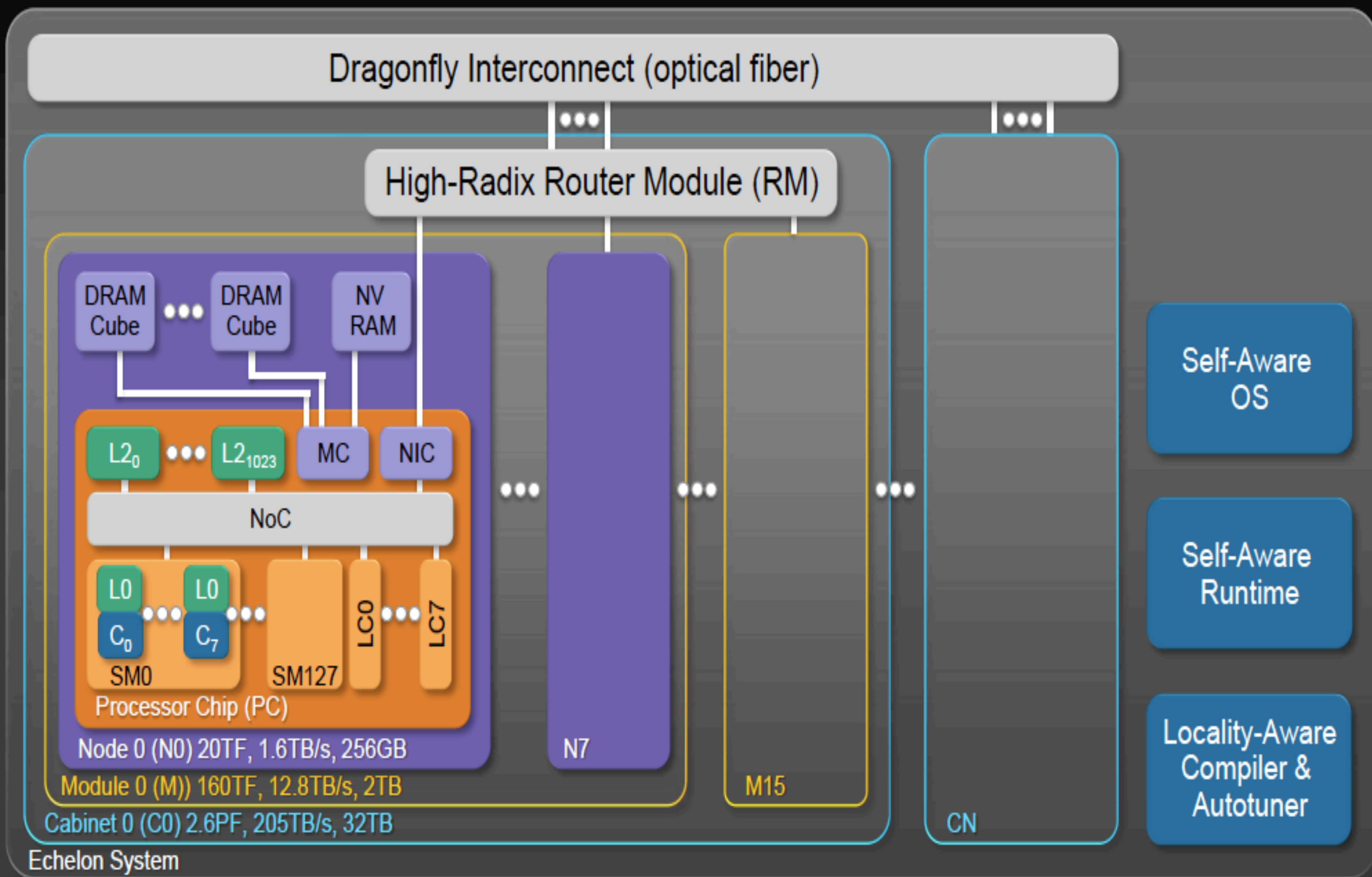
Traditional view:



(Semi-)Autotuning Compiler:



Echelon System Sketch
from "GPU Computing To Exascale and Beyond", Bill Dally, SC10



(DOE SciDAC) PERI Autotuning Tools

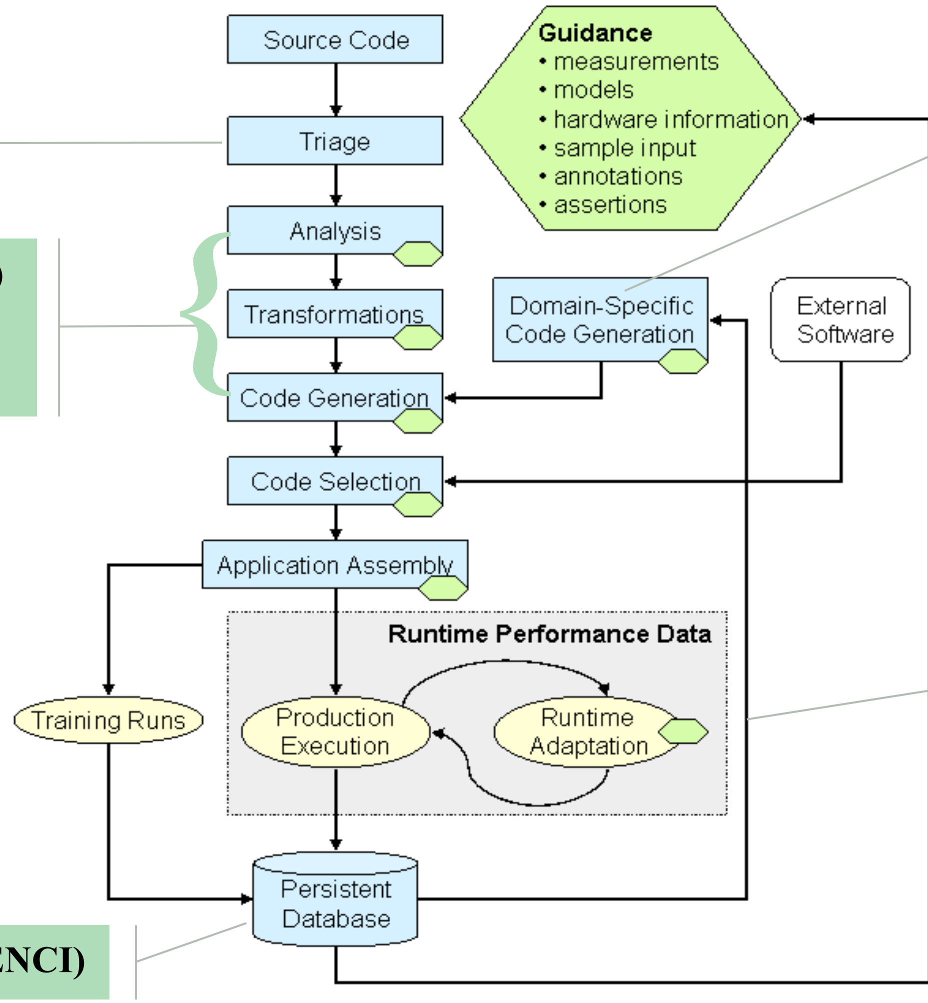
HPC Toolkit (Rice)
ROSE (LLNL)

CHiLL (USC/ISI and Utah)
ROSE (LLNL)
Orio (Argonne)

OSKI (LBNL)

Active Harmony (UMD)
GCO (UTK)

PerfTrack (LBNL, SDSC, RENCI)



Autotuning and Specialization Improve Performance of Supercomputer Applications

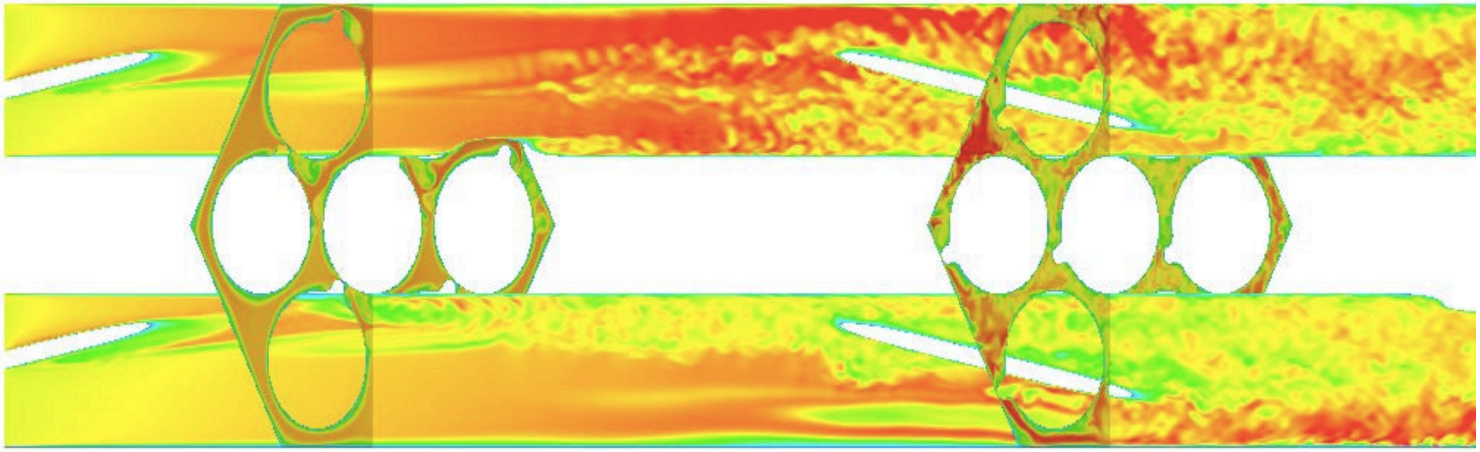
- Production scientific applications make heavy use of libraries to achieve high performance
  - E.g., BLAS, PETSc
- Using these libraries in unusual ways may be at odds with how they were optimized
- **Key Idea:** Compiler specializes library for application, and employs autotuning to achieve higher performance than manually-tuned version

Shin, Chen, Chame, Hall, and Hovland, "Autotuning and Specializing: Speeding up Matrix Multiply using Compiler Technology", IWAPT '09, Oct. 2009.

Shin, Chen, Chame, Hall, Fisher and Hovland, "Autotuning and Specializing: Speeding up Nek5000 using Compiler Technology", ICS '10, June 2010.

Autotuning of 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

THE UNIVERSITY OF UTAH

Generate Library Automatically Using CHiLL

High-level loop transformation and code generation framework
- based on polyhedral model
- script interface for programmers or compilers
- optimization strategy expressed as sequence of composable transformations

mxm.c (also supports Fortran as input)

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    for(k=0; k<n; k++)
            c[i][j]+=a[i][k]*b[k][j];
```

CHiLL

213-122.script

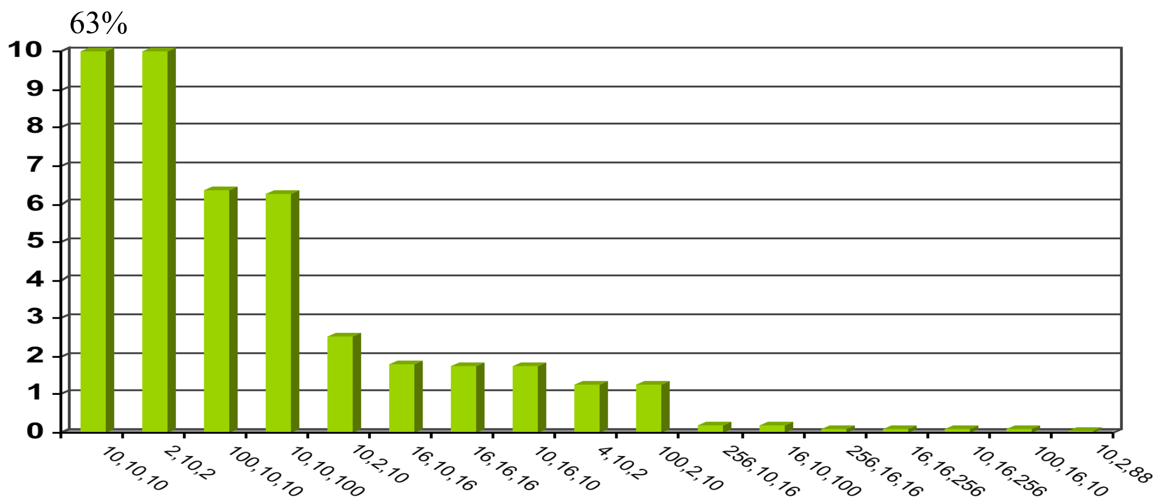
```
source: mxm.c
procedure: 0
loop: 0

known(n=10)
permute([2,1,3])
unroll(0,2,2)
unroll(0,3,2)
```

mxm213.out

```
for(j=0; j<10; j++)
  for(i=0; i<10; i+=2)
    for(k=0; k<10; k+=2){
            c[i][j]+=a[i][k]*b[k][j];
            c[i][j]+=a[i][k+1]*b[k+1][j];
            c[i+1][j]+=a[i+1][k]*b[k][j];
            c[i+1][j]+=a[i+1][k+1]*b[k+1][j];
    }
```

BLAS Library is Significant to Performance

- 8 input sizes comprise 75% of time
- Optimization opportunities
  - exploit reuse in registers (**unroll-and-jam**)
  - exploit SIMD (in the Opteron SSE-3) (**permute, unroll**)
  - reduce loop overheads (**unroll, specialize**)



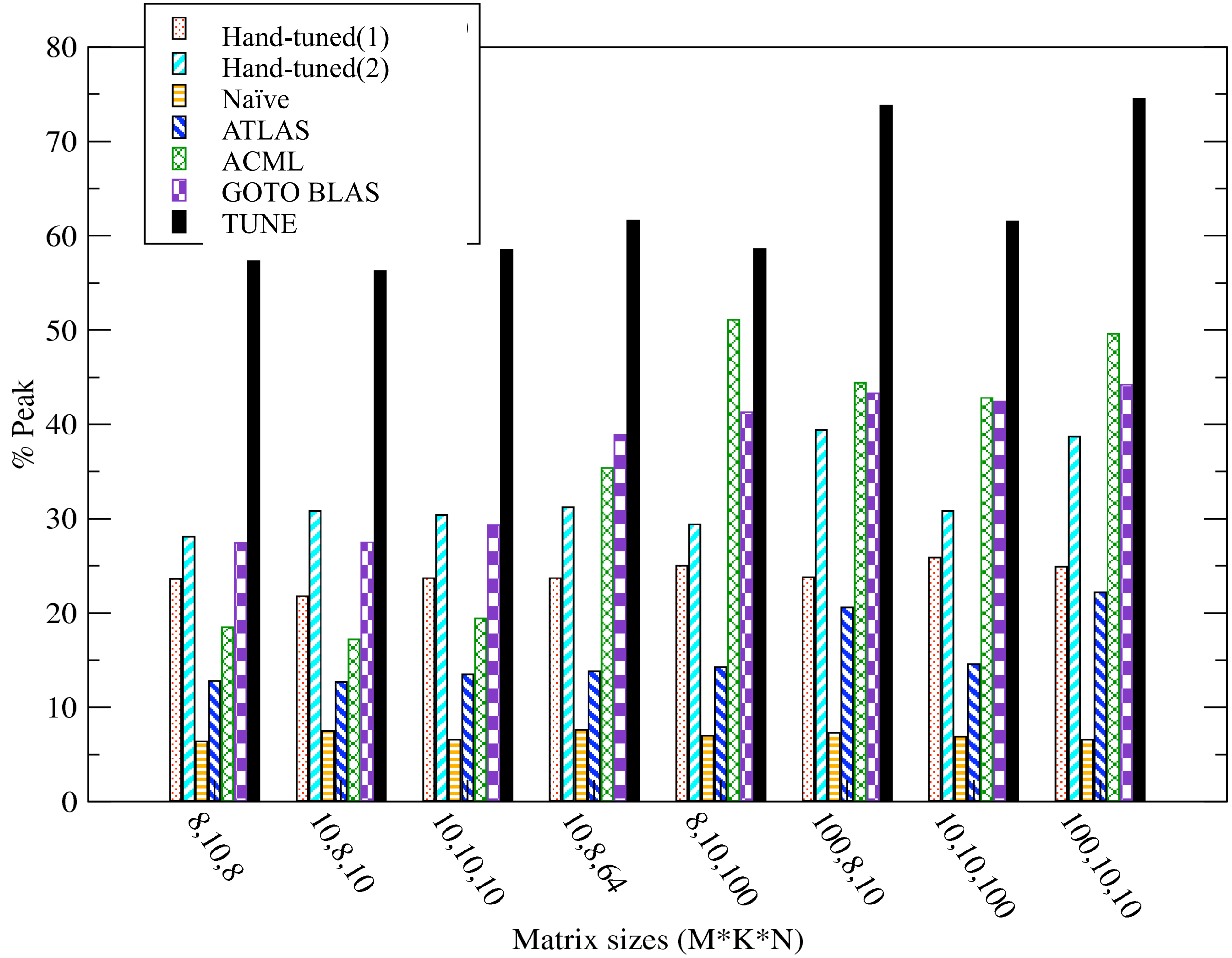
THE UNIVERSITY OF UTAH

Generated Code: Do You Want to Write This?

Example: loop order ijk, unroll 8-4-1

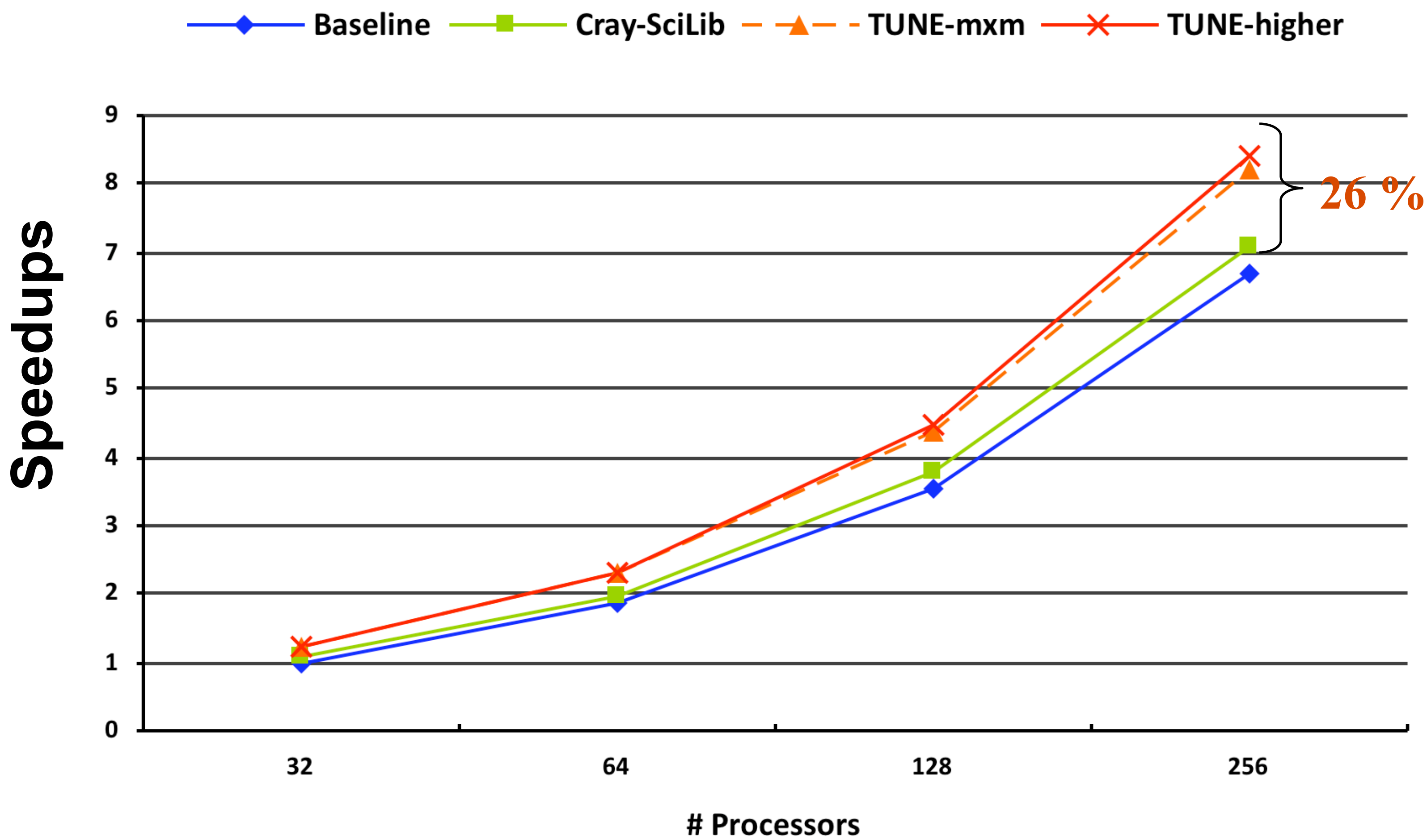
```
FUNCTION M_100_10_8 (A, B, C)

        INTEGER M_100_10_8, T4, T6
        DOUBLE PRECISION A, B, C

        DIMENSION A(8, 10)
        DIMENSION B(10, 100)
        DIMENSION C(8, 100)

        DO 2, T4 = 1, 97, 4
        C(1, T4) = 0.0000000000000000000D+00
        C(1 + 1, T4) = 0.0000000000000000000D+00
        C(1 + 2, T4) = 0.0000000000000000000D+00
        C(1 + 3, T4) = 0.0000000000000000000D+00
        C(1 + 4, T4) = 0.0000000000000000000D+00
        C(1 + 5, T4) = 0.0000000000000000000D+00
        C(1 + 6, T4) = 0.0000000000000000000D+00
        C(1 + 7, T4) = 0.0000000000000000000D+00
        C(1, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 1, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 2, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 3, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 4, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 5, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 6, T4 + 1) = 0.0000000000000000000D+00
        C(1 + 7, T4 + 1) = 0.0000000000000000000D+00
        C(1, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 1, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 2, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 3, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 4, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 5, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 6, T4 + 2) = 0.0000000000000000000D+00
        C(1 + 7, T4 + 2) = 0.0000000000000000000D+00
        C(1, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 1, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 2, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 3, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 4, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 5, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 6, T4 + 3) = 0.0000000000000000000D+00
        C(1 + 7, T4 + 3) = 0.0000000000000000000D+00

        DO 4, T6 = 1, 10, 1
            C(1, T4) = C(1, T4) + A(1, T6) * B(T6, T4)
            C(1 + 1, T4) = C(1 + 1, T4) + A(1 + 1, T6) * B(T6, T4)
            C(1 + 2, T4) = C(1 + 2, T4) + A(1 + 2, T6) * B(T6, T4)
            C(1 + 3, T4) = C(1 + 3, T4) + A(1 + 3, T6) * B(T6, T4)
            C(1 + 4, T4) = C(1 + 4, T4) + A(1 + 4, T6) * B(T6, T4)
            C(1 + 5, T4) = C(1 + 5, T4) + A(1 + 5, T6) * B(T6, T4)
            C(1 + 6, T4) = C(1 + 6, T4) + A(1 + 6, T6) * B(T6, T4)
            C(1 + 7, T4) = C(1 + 7, T4) + A(1 + 7, T6) * B(T6, T4)
            C(1, T4 + 1) = C(1, T4 + 1) + A(1, T6) * B(T6, T4 + 1)
            C(1 + 1, T4 + 1) = C(1 + 1, T4 + 1) + A(1 + 1, T6) * B(T6, T4 + 1)
            C(1 + 2, T4 + 1) = C(1 + 2, T4 + 1) + A(1 + 2, T6) * B(T6, T4 + 1)
            C(1 + 3, T4 + 1) = C(1 + 3, T4 + 1) + A(1 + 3, T6) * B(T6, T4 + 1)
            C(1 + 4, T4 + 1) = C(1 + 4, T4 + 1) + A(1 + 4, T6) * B(T6, T4 + 1)
            C(1 + 5, T4 + 1) = C(1 + 5, T4 + 1) + A(1 + 5, T6) * B(T6, T4 + 1)
            C(1 + 6, T4 + 1) = C(1 + 6, T4 + 1) + A(1 + 6, T6) * B(T6, T4 + 1)
            C(1 + 7, T4 + 1) = C(1 + 7, T4 + 1) + A(1 + 7, T6) * B(T6, T4 + 1)
            C(1, T4 + 2) = C(1, T4 + 2) + A(1, T6) * B(T6, T4 + 2)
            C(1 + 1, T4 + 2) = C(1 + 1, T4 + 2) + A(1 + 1, T6) * B(T6, T4 + 2)
            C(1 + 2, T4 + 2) = C(1 + 2, T4 + 2) + A(1 + 2, T6) * B(T6, T4 + 2)
            C(1 + 3, T4 + 2) = C(1 + 3, T4 + 2) + A(1 + 3, T6) * B(T6, T4 + 2)
            C(1 + 4, T4 + 2) = C(1 + 4, T4 + 2) + A(1 + 4, T6) * B(T6, T4 + 2)
            C(1 + 5, T4 + 2) = C(1 + 5, T4 + 2) + A(1 + 5, T6) * B(T6, T4 + 2)
            C(1 + 6, T4 + 2) = C(1 + 6, T4 + 2) + A(1 + 6, T6) * B(T6, T4 + 2)
            C(1 + 7, T4 + 2) = C(1 + 7, T4 + 2) + A(1 + 7, T6) * B(T6, T4 + 2)
            C(1, T4 + 3) = C(1, T4 + 3) + A(1, T6) * B(T6, T4 + 3)
            C(1 + 1, T4 + 3) = C(1 + 1, T4 + 3) + A(1 + 1, T6) * B(T6, T4 + 3)
            C(1 + 2, T4 + 3) = C(1 + 2, T4 + 3) + A(1 + 2, T6) * B(T6, T4 + 3)
            C(1 + 3, T4 + 3) = C(1 + 3, T4 + 3) + A(1 + 3, T6) * B(T6, T4 + 3)
            C(1 + 4, T4 + 3) = C(1 + 4, T4 + 3) + A(1 + 4, T6) * B(T6, T4 + 3)
            C(1 + 5, T4 + 3) = C(1 + 5, T4 + 3) + A(1 + 5, T6) * B(T6, T4 + 3)
            C(1 + 6, T4 + 3) = C(1 + 6, T4 + 3) + A(1 + 6, T6) * B(T6, T4 + 3)
            C(1 + 7, T4 + 3) = C(1 + 7, T4 + 3) + A(1 + 7, T6) * B(T6, T4 + 3)
        4 CONTINUE
        3 CONTINUE
        2 CONTINUE
        1 CONTINUE
          M_100_10_8 = 0
          RETURN

          END
```

Automatically Generated Code is Faster than Manually-Tuned Libraries



2.2X speedup for DGEMM
1.34X speedup for Nek5000

Nek5000 (g6a input) on Jaguar



16

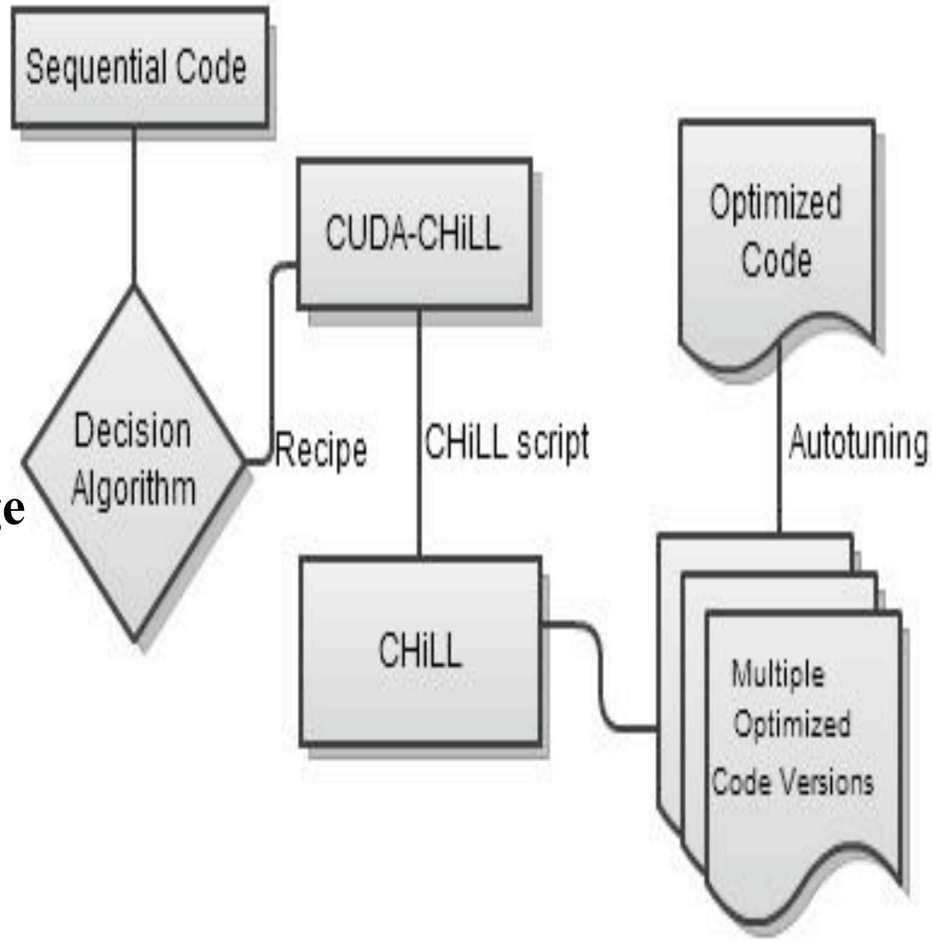
Autotuning for Future Systems

- **Future systems:** increasingly complex memory hierarchies, heterogeneous functional units
- Automatically generate CUDA for NVIDIA GPU from sequential code plus transformation recipe
- **Beyond OpenMP:** abstraction permits parallel threads & staging of data
- **Automatic parallelization:** Heuristics similar to CGO 2005 paper (Models plus heuristics plus search)
- Programming language interface provides higher levels of abstraction and permits prototyping of compiler algorithms
- **Heterogeneous code generation:** Alternative scripts generate CUDA, OpenMP or sequential code (Open CL underway) tuned for memory hierarchy

Reference: Rudy, Khan, Hall, Chen, and Chame, "A Programming Language Interface to Describe Transformations and Code Generation," LCPC '10, Oct. 2010.

CUDA-CHiLL System (UPDATE)

- **Input: Sequential C Code**
- **Optimizing decisions**
  - **Computational decomposition**
  - **Data Staging**
- **Generate transformation recipe**
  - **High-level programming language interface**
- **Polyhedral framework**
  - **Transformation and code generation**
- ***Cudaize* code**
- **Autotuning**



Guiding Transformation and Code Generation with a Transformation Recipe

Matrix-vector Multiply Source Code:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i] = a[i] + c[j][i]*b[j];
```

Example CUDA-CHiLL Recipe:

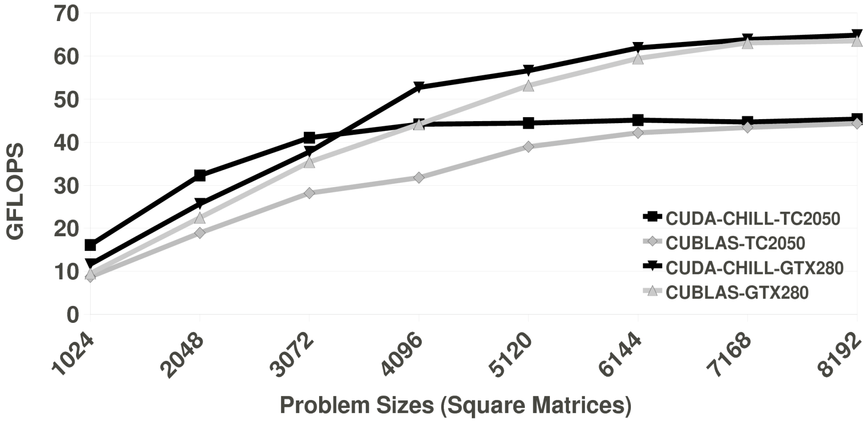
```
N = 1024
//Auto-tuning finds values for TI and
    TJ
tile_by_index({"i","j"}, {TI},
    {l1_control="ii",l2_control="jj"},
    {"ii","jj", "i", "j"})
//normalize loop to start at "i"=0
normalize_index("i")
cudaize("mv_GPU", {a=N, b=N,
    c=N*N},{block={"ii"}, thread={"i"}})
copy_to_shared("tx", "b", 1)
copy_to_registers("k", "a")
unroll_to_depth(1)
```

Automatically Generated Code:

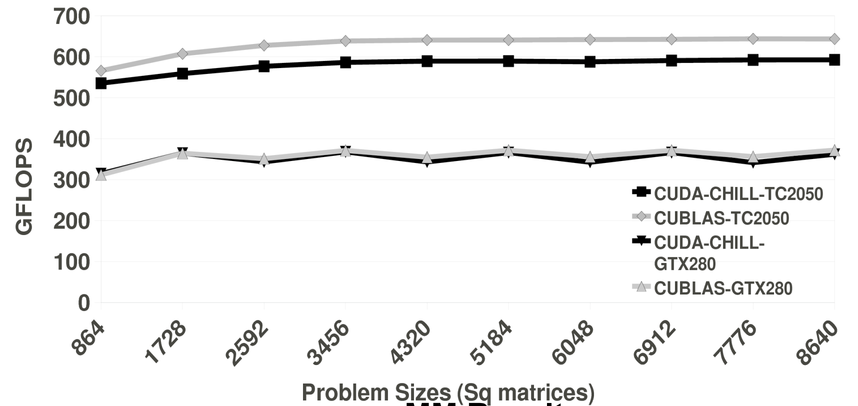
```
__global__ mv_GPU(float* a, float* b, float** c) {
  int bx = blockIdx.x; int tx = threadIdx.x;
  __shared__ float bcpy[32];
  double acpy = a[tx + 32 * bx];
  for (k = 0; k < 32; k++) {
    bcpy[tx] = b[32 * k + tx];
    __syncthreads();
    //this loop is actually fully unrolled
    for (j = 32 * k; j <= 32 * k + 32; j++) {
      acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
    }
    __synchthreads();
  }
  a[tx + 32 * bx] = acpy;
}
```

THE UNIVERSITY OF UTAH

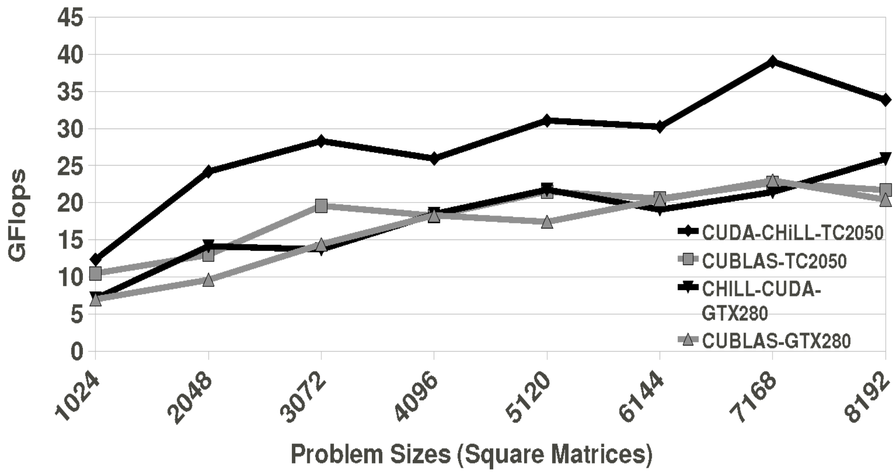
BLAS Library Kernel Optimization (SP)
Impact: Sometimes outperforms manually tuned



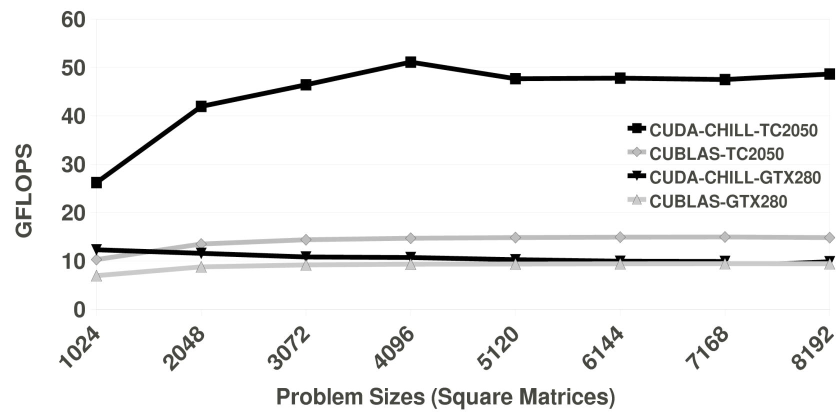
MV Results



MM Results



TMV Results



VV Results

Concluding Remarks

Compiler-based collaborative auto-tuning:

- Track manual tuning research and try to replicate (semi-) automatically
- Express mapping of software to hardware at a high level
- Explore a well-defined search space through empirical techniques
- Portable code generation, and start on heterogeneous support

Case study: linear algebra

- Code + set of transformation recipes target different architectures

Relationship to hardware design space exploration

- Systematic evaluation of many possible implementations
- Rapid evaluation and search space pruning essential