# Using CHiLL and CUDA-CHiLL
# PAID IME

Mary Hall

May 19, 2017

THE UNIVERSITY OF UTAH

# Long List of Collaborators

- Original Developer: Chun Chen
- Current Utah students:
  - Khalid Ahmad, Payal Nandy, Tharindu Rusira, Tuowen Zhao
- Ongoing collaborations:
  - LBNL: Protonu Basu, Brian van Straalen, Sam Williams
  - ANL: Prasanna Balaprakash, Hal Finkel, Paul Hovland
  - Arizona: Michelle Strout, Mahdi Mohammadi
  - Boise State: Cathie Olschanowsky
  - Intel: Anand Venkat
- Utah former students
  - Shreyas Ramalingam, Axel Rivera, Amit Roy, Gabe Rudy, Huihui Zhang
- USC students and collaborators:
  - Jacque Chame, Malik Khan, Jaewook Shin

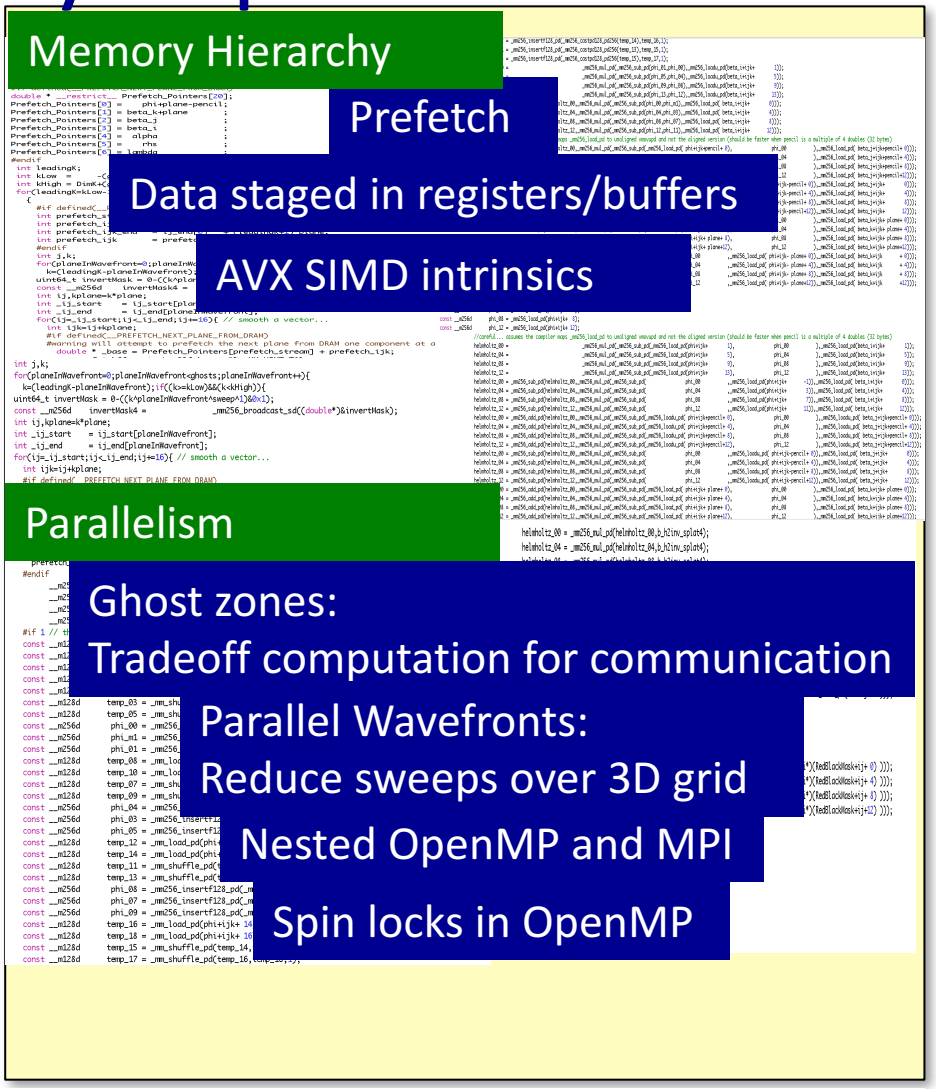
# Which version would you prefer to write?

```
/* Laplacian 7-point Variable-Coefficient Stencil */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++
      temp[k][j][i] = b * h2inv * (
        beta_i[k][j][i+1] * ( phi[k][j][i+1] – phi[k][j][i] )
            -beta_i[k][j][i] * ( phi[k][j][i] – phi[k][j][i-1] )
      +beta_j[k][j+1][i] * ( phi[k][j+1][i] – phi[k][j][i] )
      -beta_j[k][j][i] * ( phi[k][j][i] – phi[k][j-1][i] )
      +beta_k[k+1][j][i] * ( phi[k+1][j][i] – phi[k][j][i] )
      -beta_k[k][j][i] * ( phi[k][j][i] – phi[k-1][j][i] ) );
```

```
/* Helmholz */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] –
                      temp[k][j][i];
```

```
/* Gauss-Seidel Red Black Update */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++){
      if ((i+j+k+color)%2 == 0 )
      phi[k][j][i] = phi[k][j][i] – lambda[k][j][i] *
        (temp[k][j][i] – rhs[k][j][i]);}
```

**Code A:** miniGMG baseline smooth operator
approximately 13 lines of code




**Code B:** miniGMG optimized smooth operator
approximately 170 lines of code

THE UNIVERSITY OF UTAH

# And GPU Code?

**Code C:** miniGMG optimized smooth operator for GPU, 308 lines of code for just kernel

THE UNIVERSITY OF UTAH

# Code B/C is not Unusual

- Performance portability?
  - Particularly across fundamentally different CPU and GPU architectures
- Programmer productivity?
  - High performance implementations will require low-level specification that exposes architecture
- Software maintainability and portability?
  - May require multiple implementations of application

## Current solutions

- Follow MPI and OpenMP standards
  - Same code unlikely to perform well across CPU and GPU
  - Vendor C and Fortran compilers not optimized for HPC workloads
- Some domain-specific framework strategies
  - Libraries, C++ template expansion, standalone DSL
  - Not ***composable*** with other optimizations

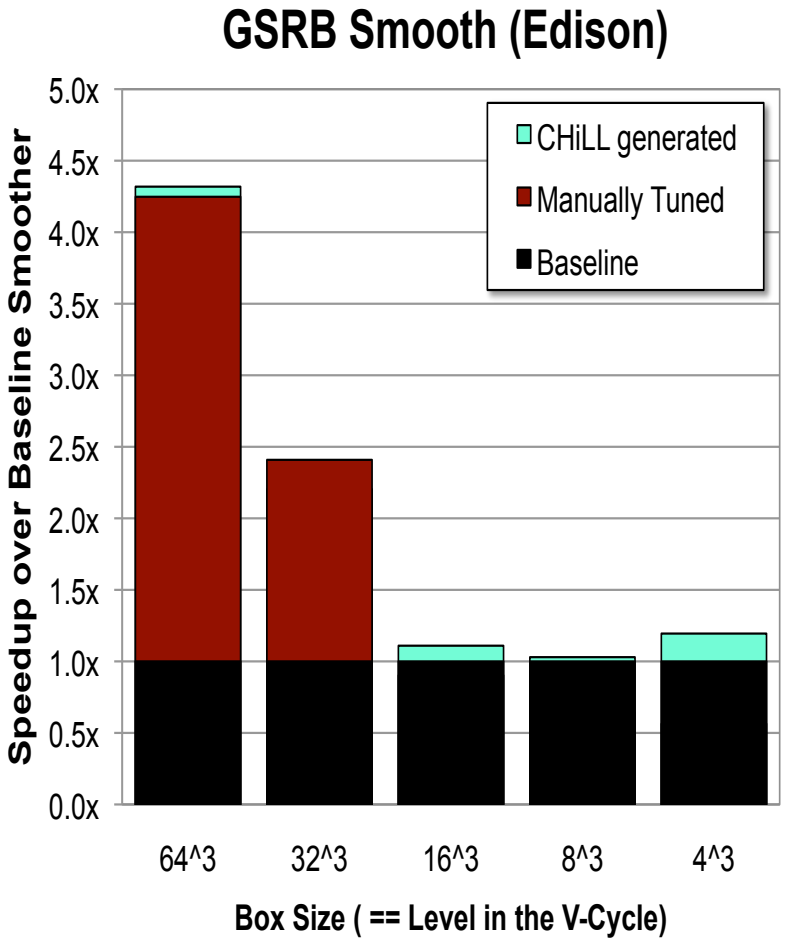
# CHiLL Approach

- CHiLL: compiler optimization framework with ***domain-specific specialization***
  - Target is loop-based scientific applications
  - ***Composable*** transformations
- Optimization strategy can be specified or derived with ***transformation recipes***
  - Also optimization parameters exposed
- ***Autotuning***
  - Systematic exploration of alternate transformation recipes and their optimization parameter values
  - Search technology to prune combinatorial space

**Automate process of generating Code B from Code A.**

# Example Result: Sometimes Code A+CHiLL beats Code B!

- miniGMG w/CHiLL
  - Fused operations
  - Communication-avoiding wavefront
  - **Parallelized (OpenMP)**
- **Autotuning** finds the best implementation for each box size
  - wavefront depth
  - nested OpenMP configuration
  - inter-thread synchronization (barrier vs. point-to-point)
- For fine grids (large arrays) CHiLL attains nearly a **4.5x speedup** over baseline




Collaborators: Protonu Basu, Sam Williams, Brian van Straalen, Lenny Oliker, Phil Colella (LBNL) [HIPC13][WOSC14][IPDPS15][PARCO17]

THE UNIVERSITY OF UTAH

# CHiLL Research

- Develop autotuning and code generation experiments with application partners
  - Stencils and multi-grid
  - Tensor contraction
  - Sparse matrix codes
- DOE Partnerships to speed up HPC applications
  - SciDAC SUPER
- PAID  (Blue Waters)
- Exascale Computing Project

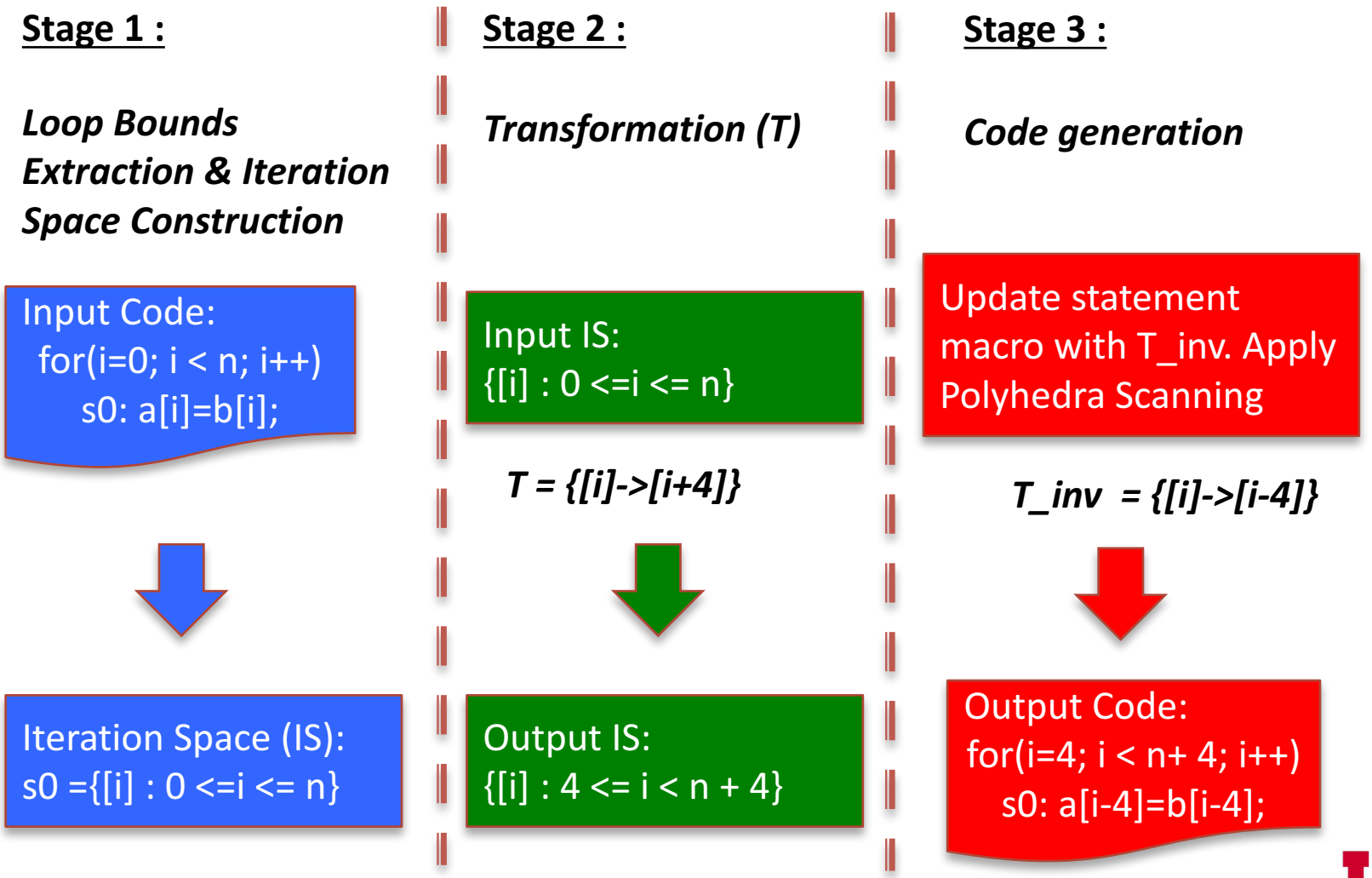
THE UNIVERSITY OF UTAH

# CHiLL in PAID

- CHiLL: Autotuning loop transformation and code generation framework
- CUDA-CHiLL: Thin layer for CUDA

| Application and PI | What we did | Impact |
|---|---|---|
| CPPTRAJ, Cheatham | Targeted GPU | 29X Speedup on key computation, Integrated into production code |
| MILC QCD, Gottlieb | Improved OpenMP Reduction | 2.4X Speedup on KNL |
| FTDT, Simpson | Thread-level Parallelism (CUDA+OpenMP) | Ongoing, 6.5X Speedup compared to sequential |

# Polyhedral Transformation and Code Generation




THE UNIVERSITY OF UTAH

# Using CHiLL

Input

Source Code:
example.c

Transformation Recipe:
example.py

***Different recipes and parameters produce different output for performance portability.***

*SuperScript encodes complete search space*


***External Search Frameworks: Active Harmony, Orio, OpenTuner***

THE UNIVERSITY OF UTAH

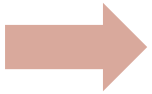
# Example CHiLL Superscripts for Code A

```
original()
skew ([0,1,2,3,4,5], 2, [3,1])
permute ([2,1,3,4])

partial_sums(0)
partial_sums(5)

fuse([0,1,2,3,4,5,6,7,8,9], 4)
```

```
omp_par_for(4,3)
```



```
original()
skew ([0,1,2,3,4,5], 2, [3,1])
permute ([2,1,3,4])

partial_sums(0)
partial_sums(5)

fuse([0,1,2,3,4,5,6,7,8,9], 4)
```

```
@begin_param_region
param(x,enum,[1,2,3,4,6,12])
param(y,enum,[1,2,3,4,6,12])
constraint(x*y==12)
omp_par_for(x,y)
@end_param_region
```

```
import chillsuper as cs
cs.generate_parameter_domain('superscript')
```

https://github.com/TharinduRusira/CHiLLsuper

# CHiLL Interactive Demos

1. Shift example
2. MILC example
3. MRIQ example (CUDA)
4. Smooth in miniGMG

# MILC and CHiLL

- Dslash operator dominates execution
- Different data layout/implementation for each new architecture, architecture-specific

```
for (it1=0; it1<4; it1++) {
  for (it2=0; it2<3; it2++) {
    for (it3=0; it3<3; it3++) {
      tc_real[it3] += ta_real[it1][it3[it2] *
     tb_real[it1][it2];
      tc_imag[it3] += ta_real[it1][it3][it2]
     * tb_imag[it1][it2];
      tc_real[it3] -= ta_imag[it1][it3][it2]
     * tb_imag[it1][it2];
      tc_imag[it3] +=
ta_imag[it1][it3][it2]      *
tb_real[it1][it2];
 } }}
```

Code A: 3x3 complex matrix-vector multiply
7 lines of code.

```
for(n=0;n<4;n++,mat++){
 switch(n){
 case(0): b=b0; break;  case(1): b=b1; break;   case(2): b=b2; break;
 case(3): b=b3; break;   default: b = 0;   }
 br=b->c[0].real;   bi=b->c[0].imag;
 a0=mat->e[0][0].real;   a1=mat->e[1][0].real;   a2=mat->e[2][0].real;
 c0r += a0*br;   c1r += a1*br;   c2r += a2*br;
 c0i += a0*bi;   c1i += a1*bi;   c2i += a2*bi;
 a0=mat->e[0][0].imag;   a1=mat->e[1][0].imag;   a2=mat->e[2][0].imag;
 c0r -= a0*bi;   c1r -= a1*bi;   c2r -= a2*bi;
 c0i += a0*br;   c1i += a1*br;   c2i += a2*br;
 br=b->c[1].real;  bi=b->c[1].imag;  a0=mat->e[0][1].real;   a1=mat->e[1][1].real;
 c0r += a0*br;   c1r += a1*br;   c2r += a2*br;
 c0i += a0*bi;   c1i += a1*bi;   c2i += a2*bi;
 a0=mat->e[0][1].imag;   a1=mat->e[1][1].imag;   a2=mat->e[2][1].imag;
 c0r -= a0*bi;   c1r -= a1*bi;   c2r -= a2*bi;
 c0i += a0*br;   c1i += a1*br;   c2i += a2*br;
  br=b->c[2].real;      bi=b->c[2].imag;   a0=mat->e[0][2].real;   a1=mat-
>e[1][2].real;   a2=mat->e[2][2].real;   c0r += a0*br;   c1r += a1*br;   c2r +=
a2*br;   c0i += a0*bi;   c1i += a1*bi;   c2i += a2*bi;   a0=mat->e[0][2].imag;
a1=mat->e[1][2].imag;   a2=mat->e[2][2].imag;   c0r -= a0*bi;   c1r -= a1*bi;
c2r -= a2*bi;   c0i += a0*br;   c1i += a1*br;   c2i += a2*br;
} c->c[0].real = c0r;  c->c[0].imag = c0i;  c->c[1].real = c1r;  c->c[1].imag = c1i;
  c->c[2].real = c2r;  c->c[2].imag = c2i;}
c->c[0].real = c0r;  c->c[0].imag = c0i;  c->c[1].real = c1r;  c->c[1].imag = c1i;
c->c[2].real = c2r;  c->c[2].imag = c2i;}
```

Code B: Example implementation in MILC
72 lines of code.

THE UNIVERSITY OF UTAH

# MRIQ Optimized Example

```
1   permute(0,{"j","i"})
2   tile_by_index({"j"},{TI},{l1_control="jj"},{"jj","j","i"})
3   normalize_index("j")
4   normalize_index("i")
5   tile_by_index({"i"},{TJ},{l1_control="ii",l1_tile="i"},
                    {"jj","ii","j","i"})
6   cudaize("kernel_GPU",{x=N,y=N,z=N,Qr=N,Qi=N,kVals=M},
              {block={"jj"},thread={"j"}})
5   copy_to_shared("tx","kVals",1)
6   copy_to_texture("kVals)
7   copy_to_registers("ii","x")
8   copy_to_registers("ii","y")
9   copy_to_registers("ii","z")
10 copy_to_registers("ii","Qi")
11 copy_to_registers("ii","Qr")
```

(Output: See file)

From Khan et al., A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim. 9, 4, Article 31 (January 2013).*

# Conclusion

- Autotuning compiler technology useful to achieve performance portability on new architectures!

- First time working with a facilities team
- Software improvements
  - Test cases, doxygen, github, updates to manual
  - Abstraction that separates from compiler AST
- New Research
  - Data layout optimization (MILC)
  - C++ iterators and vector library support (CPPTRAJ)
  - Specialization of coefficient matrix (FTDT)