

# Locus: A System and a Language for Program Optimization

Thiago Teixeira\*, Corinne Ancourt+, David Padua\*, William Gropp\*

\*Department of Computer Science, University of Illinois at Urbana-Champaign, USA

+MINES ParisTech, PSL University, France



CGO - Washington, DC - Feb 2019

# Introduction



# Introduction

- Very complex machines
- Gap between performance of hand-tuned and compiler-generated code has grown substantially

# Introduction

- Very complex machines
- Gap between performance of hand-tuned and compiler-generated code has grown substantially
- Platform-specific optimizations are required



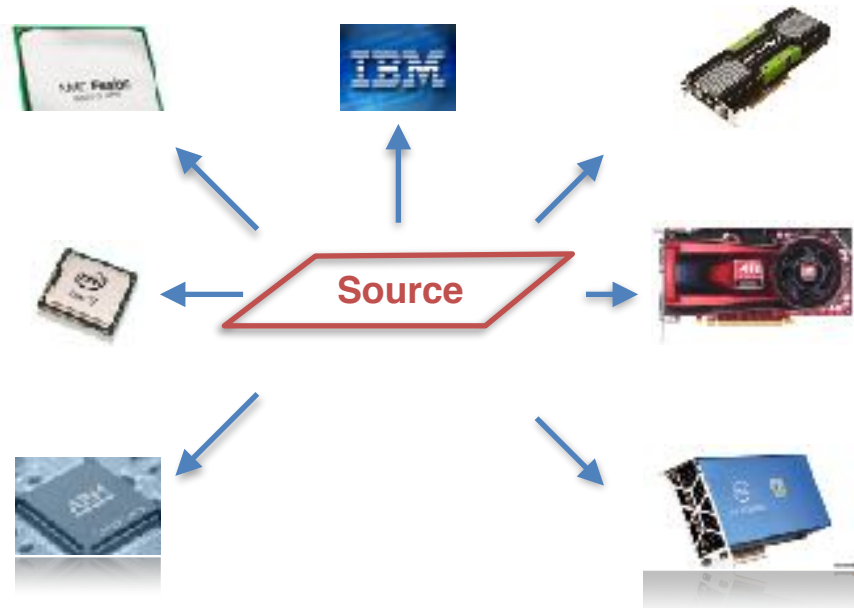
# Introduction

- Very complex machines
- Gap between performance of hand-tuned and compiler-generated code has grown substantially
- Platform-specific optimizations are required
- Platforms change, and new ones are introduced



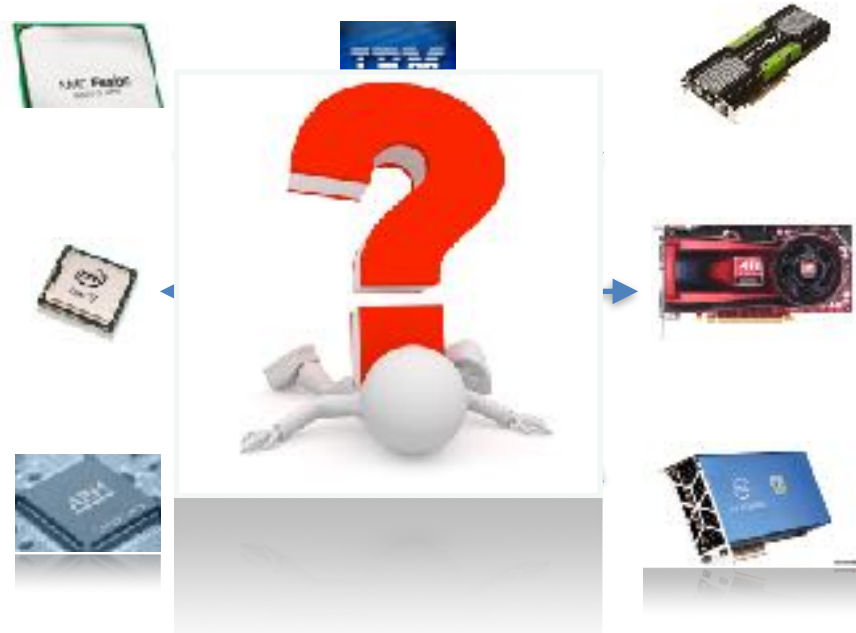
# Introduction

- Very complex machines
- Gap between performance of hand-tuned and compiler-generated code has grown substantially
- Platform-specific optimizations are required
- Platforms change, and new ones are introduced



# Introduction

- Very complex machines
- Gap between performance of hand-tuned and compiler-generated code has grown substantially
- Platform-specific optimizations are required
- Platforms change, and new ones are introduced
- As you add them the code becomes less and less maintainable and understandable



# Goal

- Improve performance automatically
- Target multiple platforms
- Keep the code maintainable in the long term





# Goal

- Improve performance automatically
- Target multiple platforms
- Keep the code maintainable in the long term
- How?



# Goal

- Improve performance automatically
- Target multiple platforms
- Keep the code maintainable in the long term
- How?

Automatically generate and evaluate a collection of optimized variants  
by executing them



# Challenges

# Challenges

1. How to describe a collection of optimized variants (opt space) concisely?
  - modify and extend the use of optimizations

# Challenges

1. How to describe a collection of optimized variants (opt space) concisely?
  - modify and extend the use of optimizations
2. Generate the variants automatically:
  - often needs multiple techniques
  - a lot tools out there
  - tools are not prepared to work with each other
  - compose a diverse set of transformations into a final code is not trivial



# Challenges

1. How to describe a collection of optimized variants (opt space) concisely?
  - modify and extend the use of optimizations
2. Generate the variants automatically:
  - often needs multiple techniques
  - a lot tools out there
  - tools are not prepared to work with each other
  - compose a diverse set of transformations into a final code is not trivial
3. Select relevant variants
  - optimization space too large to be fully evaluated

# Challenges

1. How to describe a collection of optimized variants (opt space) concisely?
  - modify and extend the use of optimizations
2. Generate the variants automatically:
  - often needs multiple techniques
  - a lot tools out there
  - tools are not prepared to work with each other
  - compose a diverse set of transformations into a final code is not trivial
3. Select relevant variants
  - optimization space too large to be fully evaluated
4. Manage platform-specific recipes of transformations
  - how and where to store
  - make it available to non-experts



# Optimization Space

- triple nested loop

```
for i
  for j
    for k
```





# Optimization Space

- triple nested loop

6 variants

```
for i
  for j
    for k
```

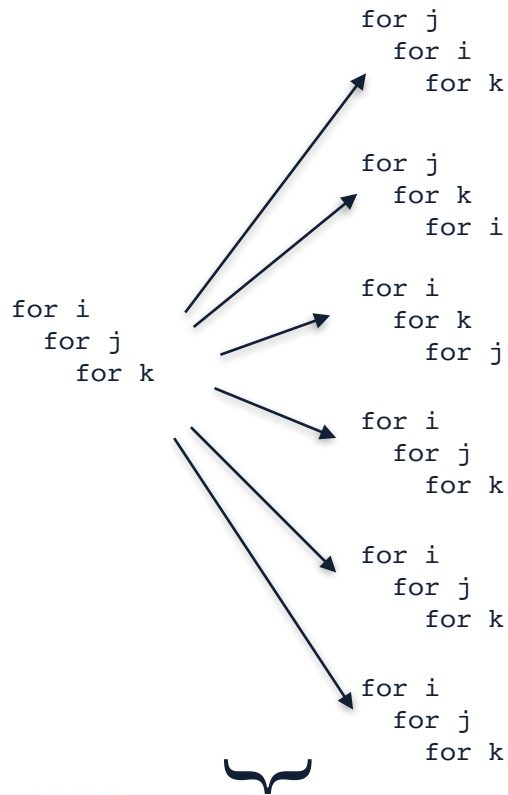


Interchange  
all permutations

# Optimization Space

- triple nested loop

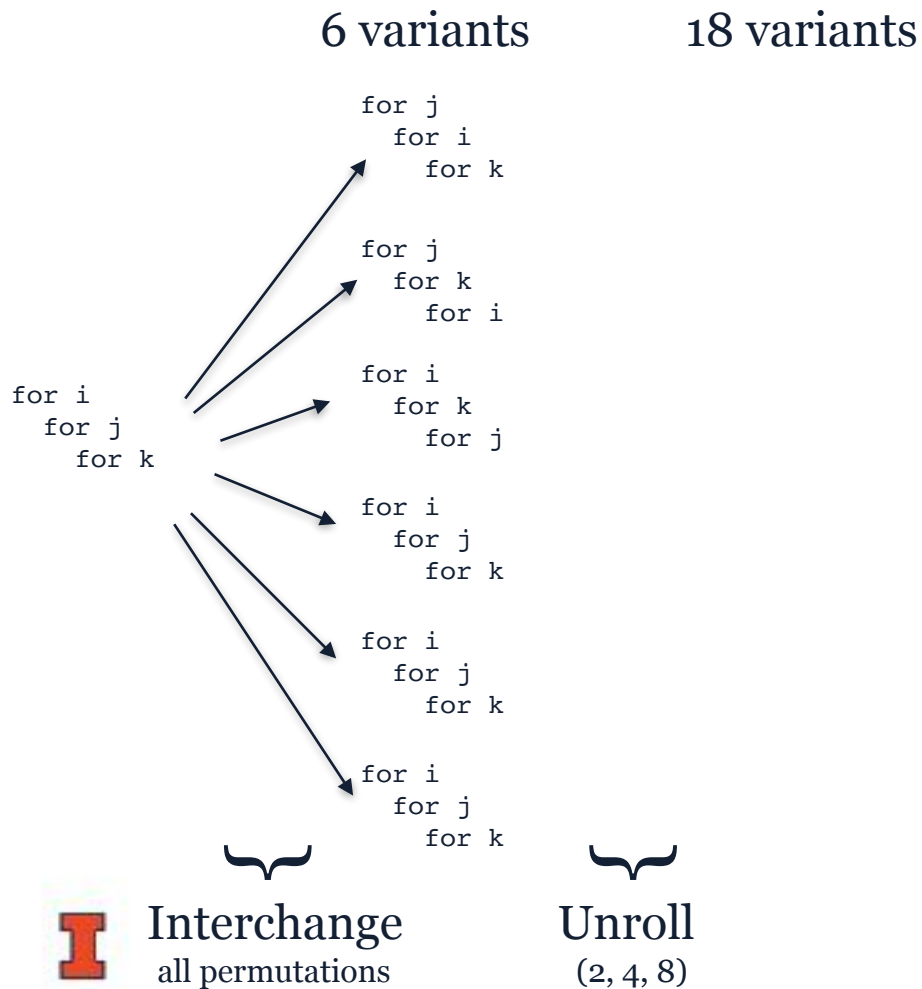
6 variants



Interchange  
all permutations

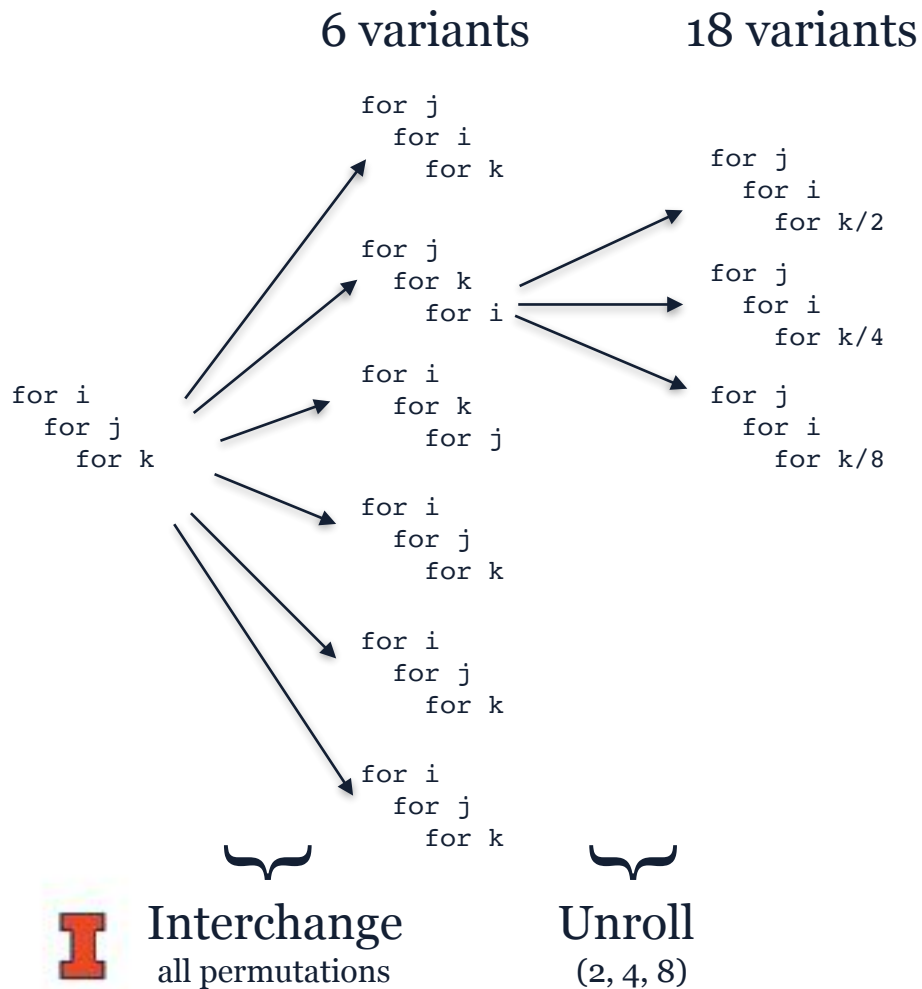
# Optimization Space

- triple nested loop



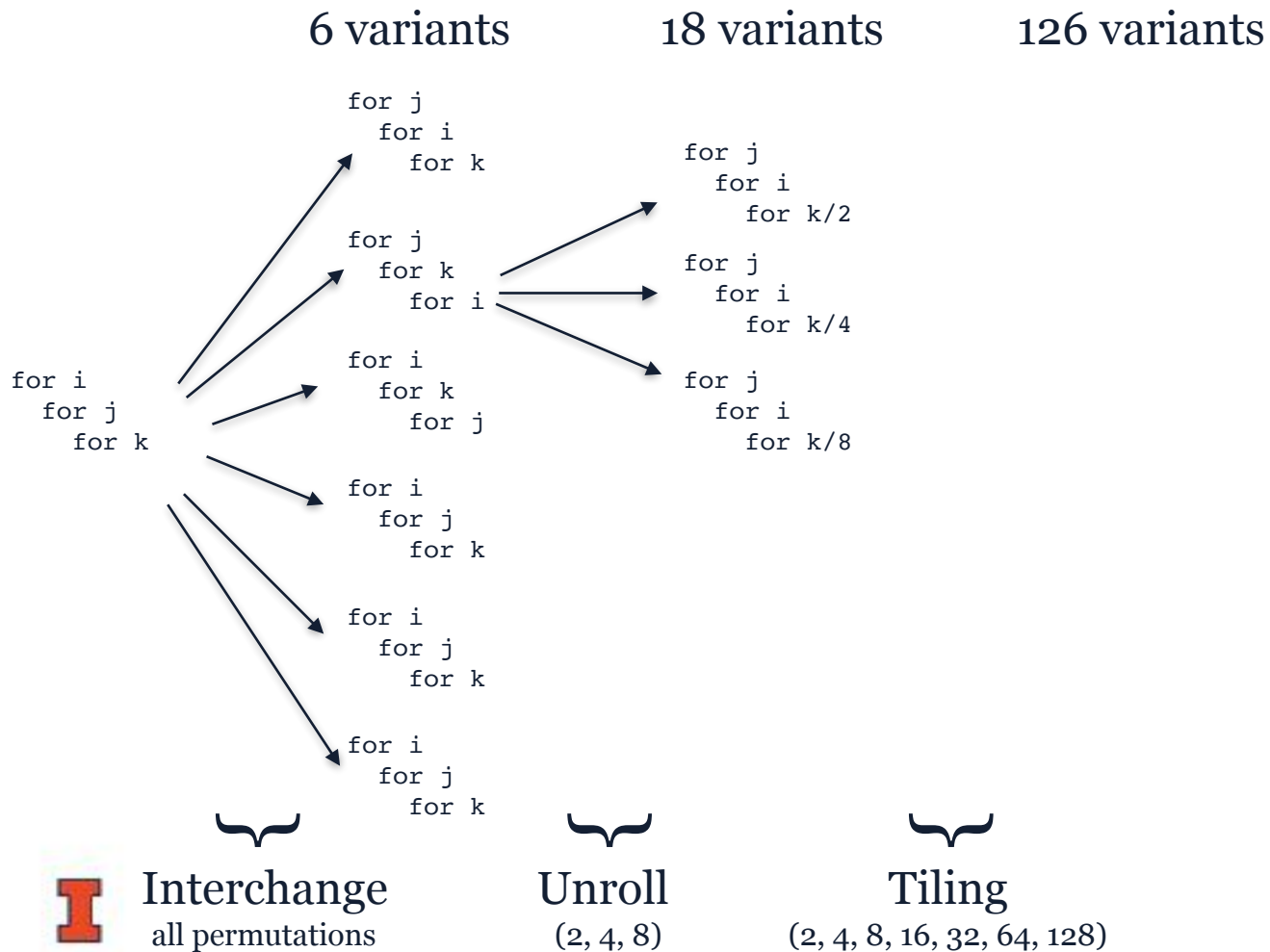
# Optimization Space

- triple nested loop



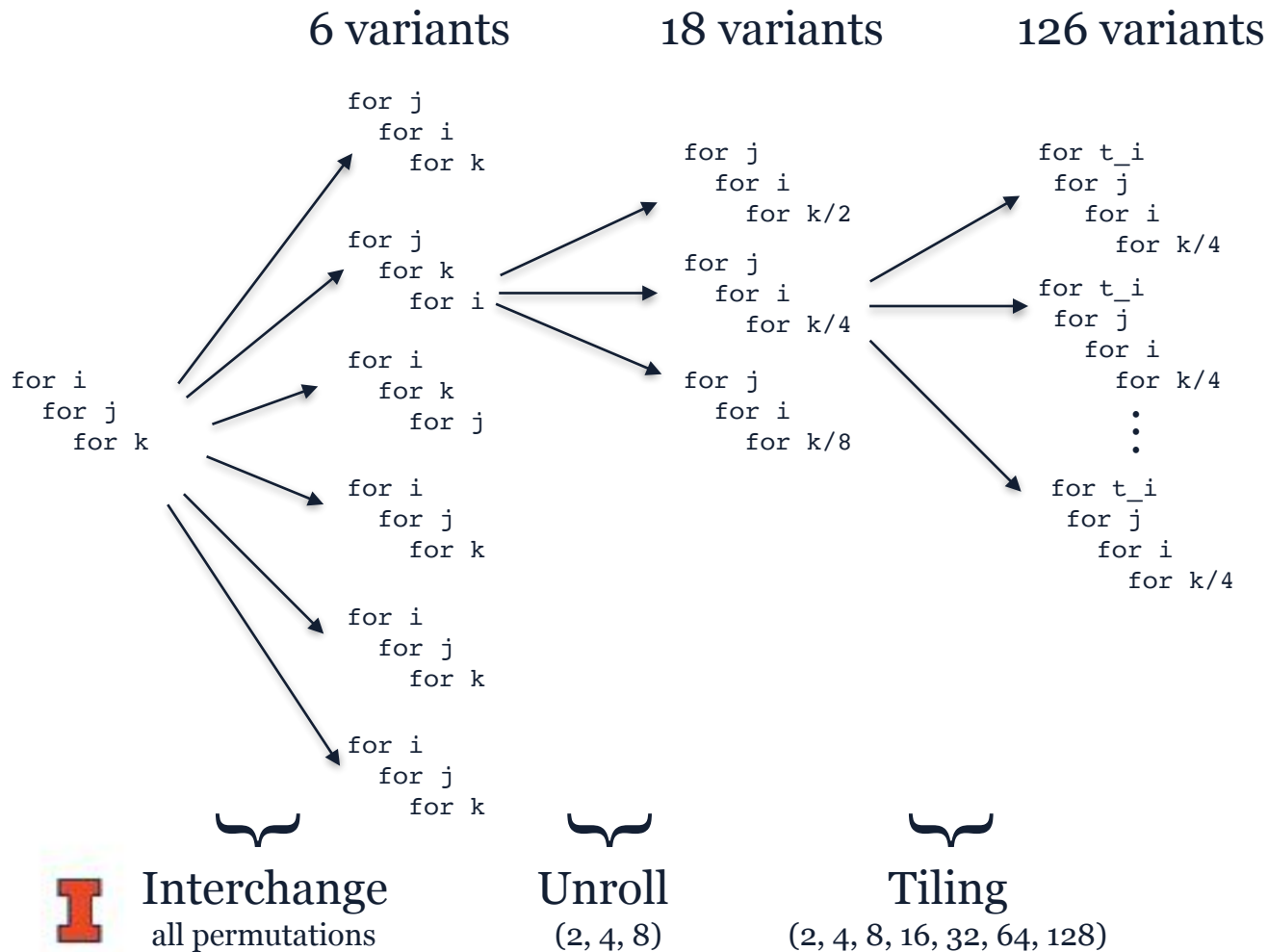
# Optimization Space

- triple nested loop



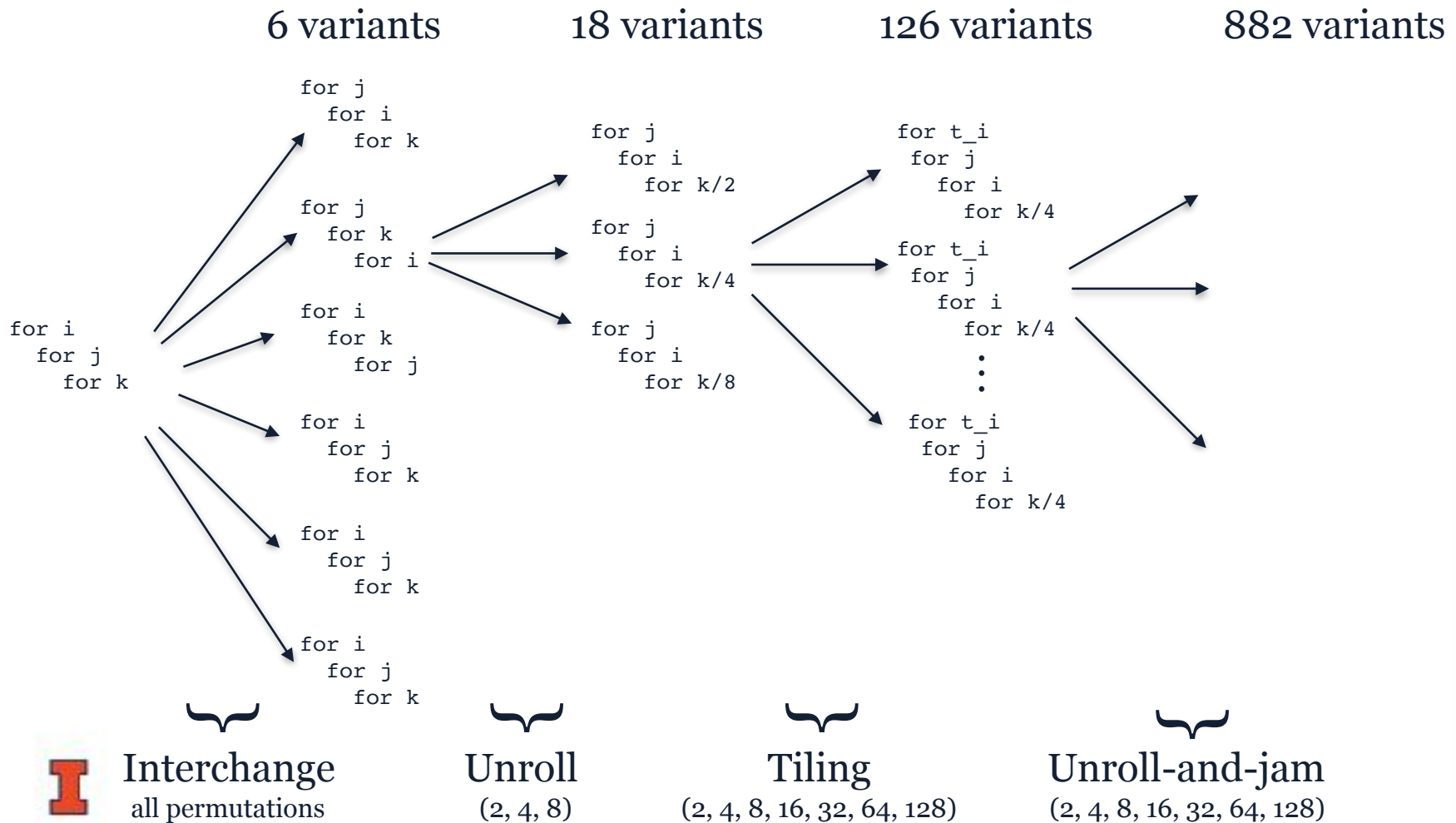
# Optimization Space

- triple nested loop

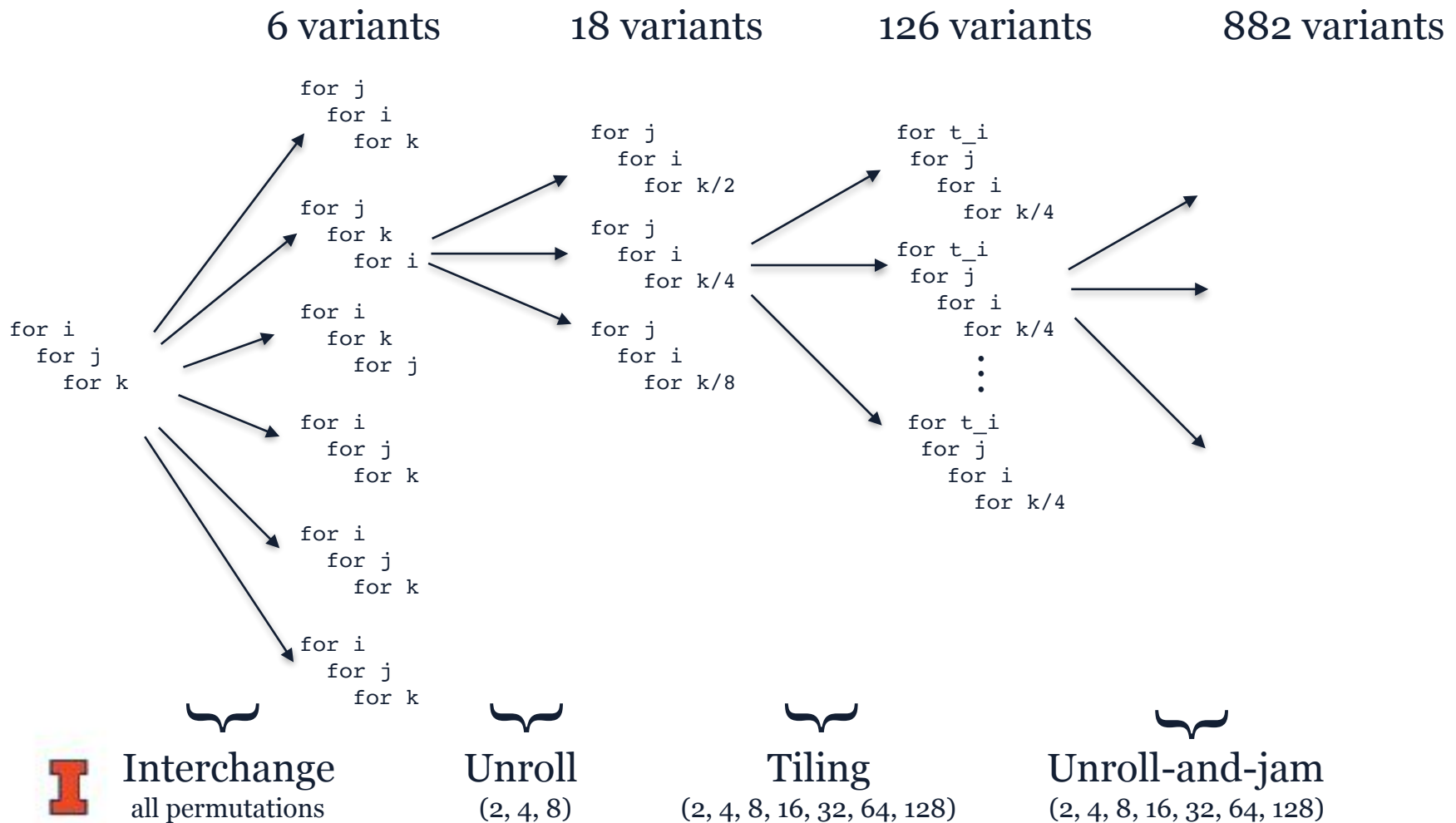


# Optimization Space

- triple nested loop



# Locus





# Locus

Locus program

```
for i
  for j
    for k
```

 + 

# Locus

Locus program

```
for i
  for j
    for k
```

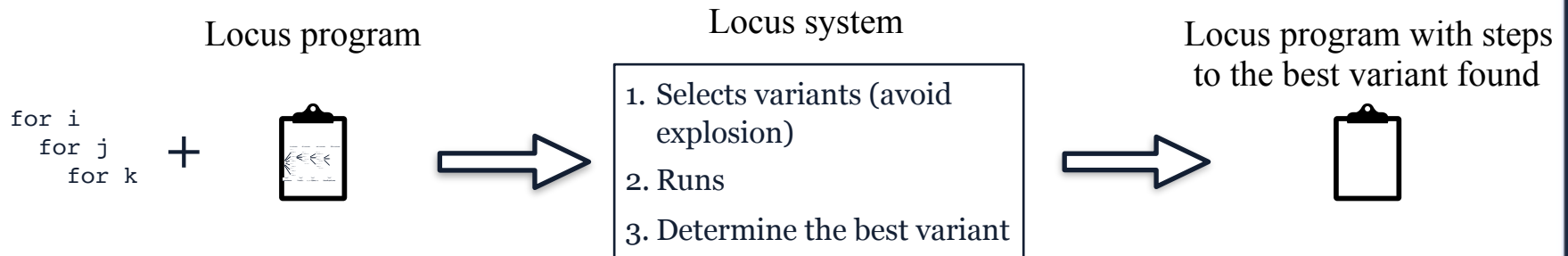
+



Locus system

1. Selects variants (avoid explosion)
2. Runs
3. Determine the best variant

# Locus



# Locus

- Semi-automatic approach to assist performance experts and code developers in the performance optimization of programs in C, C++, and Fortran
- Orchestrates the application of transformations to a baseline version of the code
- Specially for optimizing complex, long-lived applications running on different environments



# Contributions

# Contributions

- Defined Locus language:
  - describe *concisely* complex space of optimizations
  - *agnostic* of any specific traversal method
  - *decouple* performance expert role from application expert role

# Contributions

- Defined Locus language:
  - describe *concisely* complex space of optimizations
  - *agnostic* of any specific traversal method
  - *decouple* performance expert role from application expert role
- Implemented a system with flexible API for plugging in:
  - *different* variant selection techniques (optimization space traversal)
  - *collection* of transformations developed internally and externally



# Contributions

- Defined Locus language:
  - describe *concisely* complex space of optimizations
  - *agnostic* of any specific traversal method
  - *decouple* performance expert role from application expert role
- Implemented a system with flexible API for plugging in:
  - *different* variant selection techniques (optimization space traversal)
  - *collection* of transformations developed internally and externally
- Optimizer and interpreter for the Locus programs:
  - *prune* the space automatically
  - speeds-up the empirical search





# Locus Approach

- Baseline code: defined by the developer, no platform- or compiler-specific optimizations
- Annotated regions of interest (i.e., code regions)
- Program the application of the optimizations for each code region



# Locus System

## Annotated Source Code

```
#pragma @Locus loop = matmul
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<K; k++)
      C[i][j] = beta*C[i][j]
        + alpha*A[i][k]*B[k][j];
```



# Locus System

## Annotated Source Code

```
#pragma @Locus loop = matmul
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<K; k++)
      C[i][j] = beta*C[i][j]
               + alpha*A[i][k]*B[k][j];
```



# Locus System

## Annotated Source Code

```
#pragma @Locus loop = matmul
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<K; k++)
      C[i][j] = beta*C[i][j]
        + alpha*A[i][k]*B[k][j];
```

## Locus Program

```
CodeReg matmul {
  tiledim = 4;
  tiletype = Tiling2D() OR Tiling3D();
  printstatus(tiletype);
  if (tiletype == "2D") {
    RoseLocus.Unroll(loop=innermost, factor=tiledim);
  }
}
```



# Locus System

## Annotated Source Code

```
#pragma @Locus loop = matmul
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<K; k++)
      C[i][j] = beta*C[i][j]
        + alpha*A[i][k]*B[k][j];
```

## Locus Program

```
CodeReg matmul {
  tiledim = 4;
  tiletype = Tiling2D() OR Tiling3D();
  printstatus(tiletype);
  if (tiletype == "2D") {
    RoseLocus.Unroll(loop=innermost, factor=tiledim);
  }
}
```



# Locus System

## Annotated Source Code

```
#pragma @Locus loop = matmul
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<K; k++)
      C[i][j] = beta*C[i][j]
        + alpha*A[i][k]*B[k][j];
```

## Locus Program

```
CodeReg matmul {
  tiledim = 4;
  tiletype = Tiling2D() OR Tiling3D();
  printstatus(tiletype);
  if (tiletype == "2D") {
    RoseLocus.Unroll(loop=innermost, factor=tiledim);
  }
}
```

- Optimizations are target-specific and region-specific
- Separated from the application's code



# Locus Optimization Language



# Locus Optimization Language

- Optimization recipes for each code region (CodeReg, OptSeq)



# Locus Optimization Language

- Optimization recipes for each code region (CodeReg, OptSeq)
- Loops, If-then-else

# Locus Optimization Language

- Optimization recipes for each code region (CodeReg, OptSeq)
- Loops, If-then-else
- Special Search Constructs:
  - OR blocks and statements;
  - Optional statements;
  - *enum, integer, permutation, poweroftwo...*



# Locus Optimization Language

Interchange



Tiling

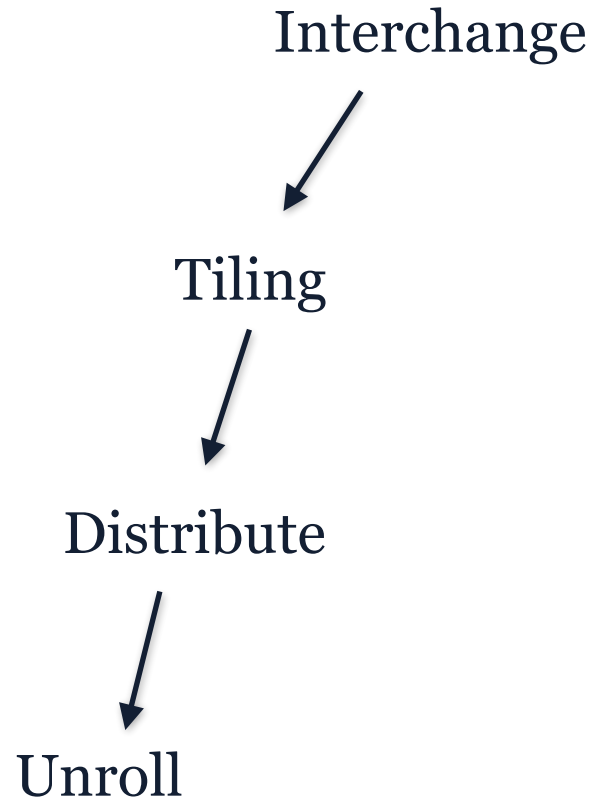


Distribute

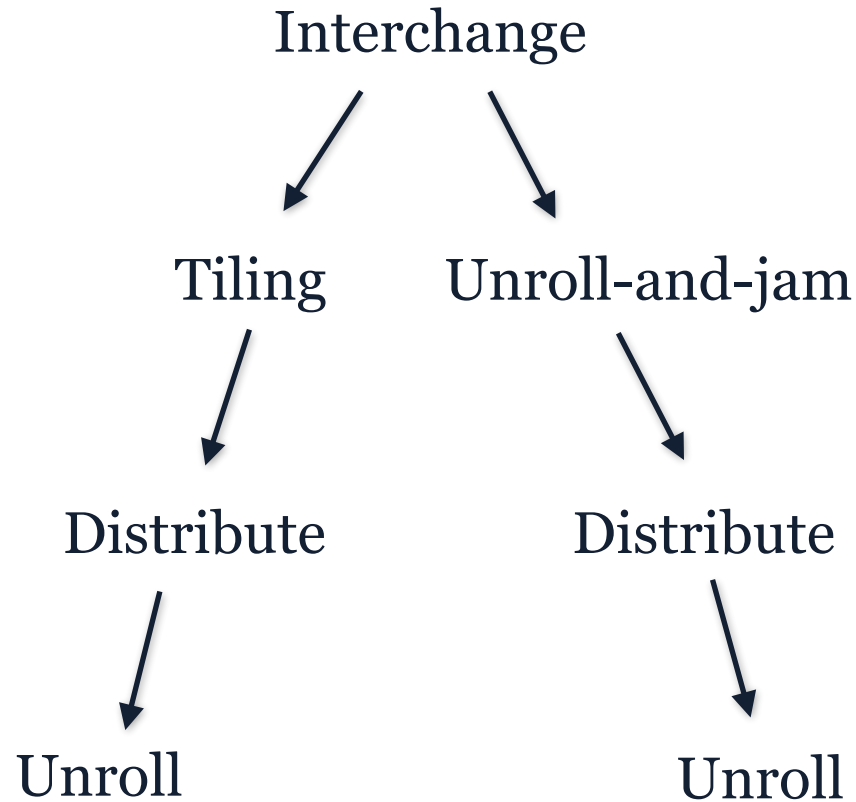


Unroll

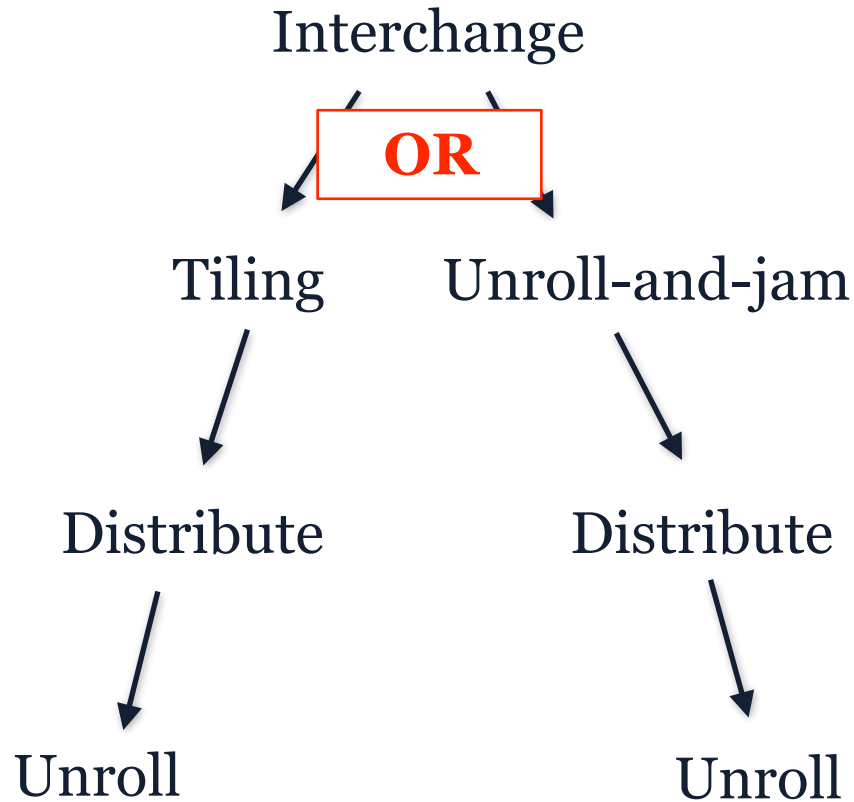
# Locus Optimization Language



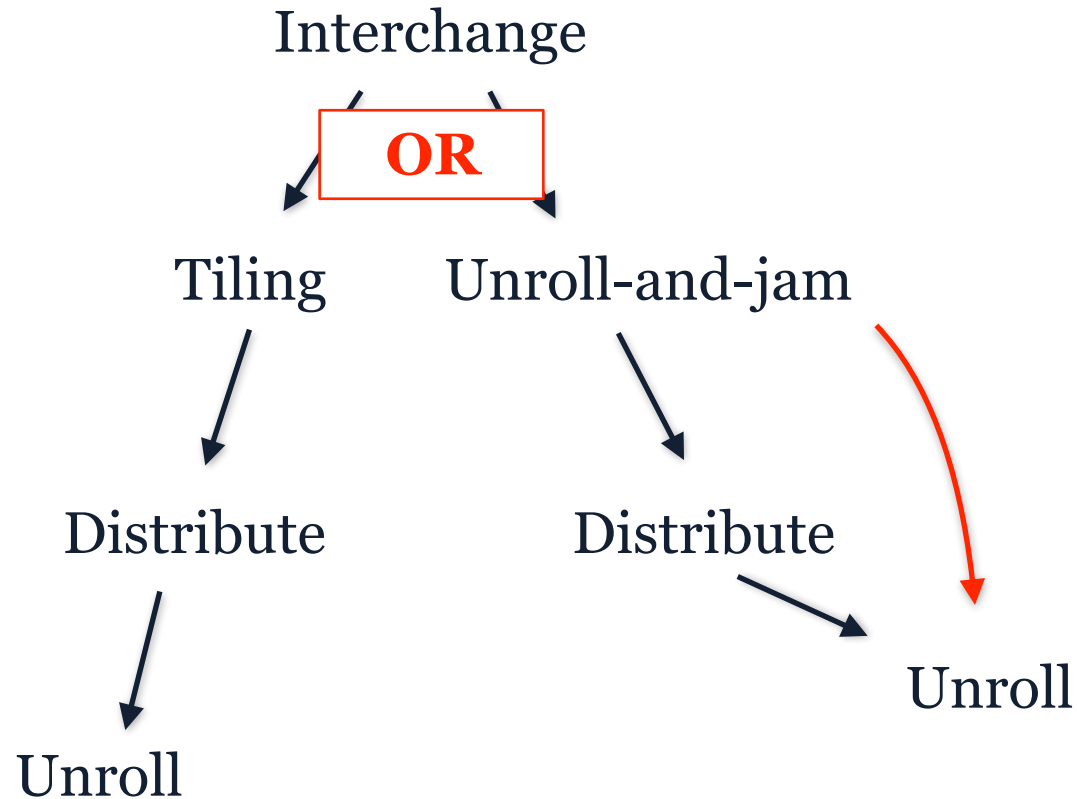
# Locus Optimization Language



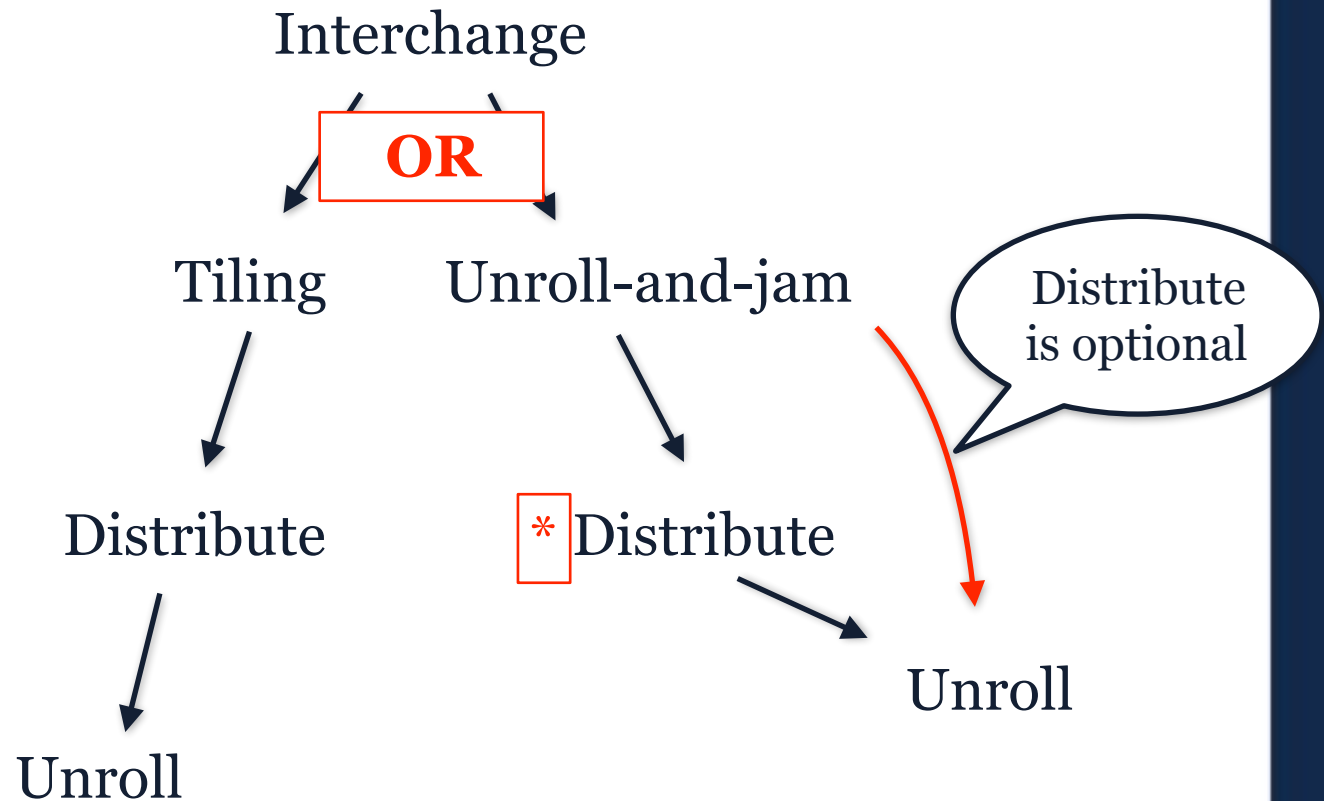
# Locus Optimization Language



# Locus Optimization Language



# Locus Optimization Language





# Modules Integration <sup>1/3</sup>



# Modules Integration <sup>1/3</sup>

- Locus defines an entire search space



# Modules Integration <sup>1/3</sup>

- Locus defines an entire search space
- Given the search space, one must:
  - decide which variants to evaluate (search module)
  - use tools to generate each variant's transformation plan (transformation module)

# Modules Integration <sup>1/3</sup>

- Locus defines an entire search space
- Given the search space, one must:
  - decide which variants to evaluate (search module)
  - use tools to generate each variant's transformation plan (transformation module)
- Locus allows for both multiple search and transformation modules



# Modules Integration <sup>1/3</sup>

- Locus defines an entire search space
- Given the search space, one must:
  - decide which variants to evaluate (search module)
  - use tools to generate each variant's transformation plan (transformation module)
- Locus allows for both multiple search and transformation modules
- Collaborative environment, reuse other's work



# Modules Integration <sup>2/3</sup>



# Modules Integration <sup>2/3</sup>

- Search modules (OpenTuner, HyperOpt):



# Modules Integration <sup>2/3</sup>

- Search modules (OpenTuner, HyperOpt):
  - Convert the Locus' space to module's space



# Modules Integration <sup>2/3</sup>

- Search modules (OpenTuner, HyperOpt):
  - Convert the Locus' space to module's space
    - parameters, OR statements and blocks, conditionals

# Modules Integration <sup>2/3</sup>

- Search modules (OpenTuner, HyperOpt):
  - Convert the Locus' space to module's space
    - parameters, OR statements and blocks, conditionals
  - Search: start process



# Modules Integration <sup>2/3</sup>

- Search modules (OpenTuner, HyperOpt):
  - Convert the Locus' space to module's space
    - parameters, OR statements and blocks, conditionals
  - Search: start process
  - For each point converts it back to Locus representation, and invokes the interpreter



# Modules Integration <sup>3/3</sup>



# Modules Integration <sup>3/3</sup>

- Transformation modules (Pips, RoseLocus, Pragmas, BuiltIn):
  - Allows for fine-grain selection
    - Can pick a different module for each transformation (e.g., Interchange, Tiling)
  - Optimizations on code region level
  - Workflow:
    - Locus transforms to modules notation
    - Module applies the optimization
    - Locus transforms the resulting code into its internal representation (AST and code region structure)
  - It has shown to be flexible enough to integrate other transformations if needed

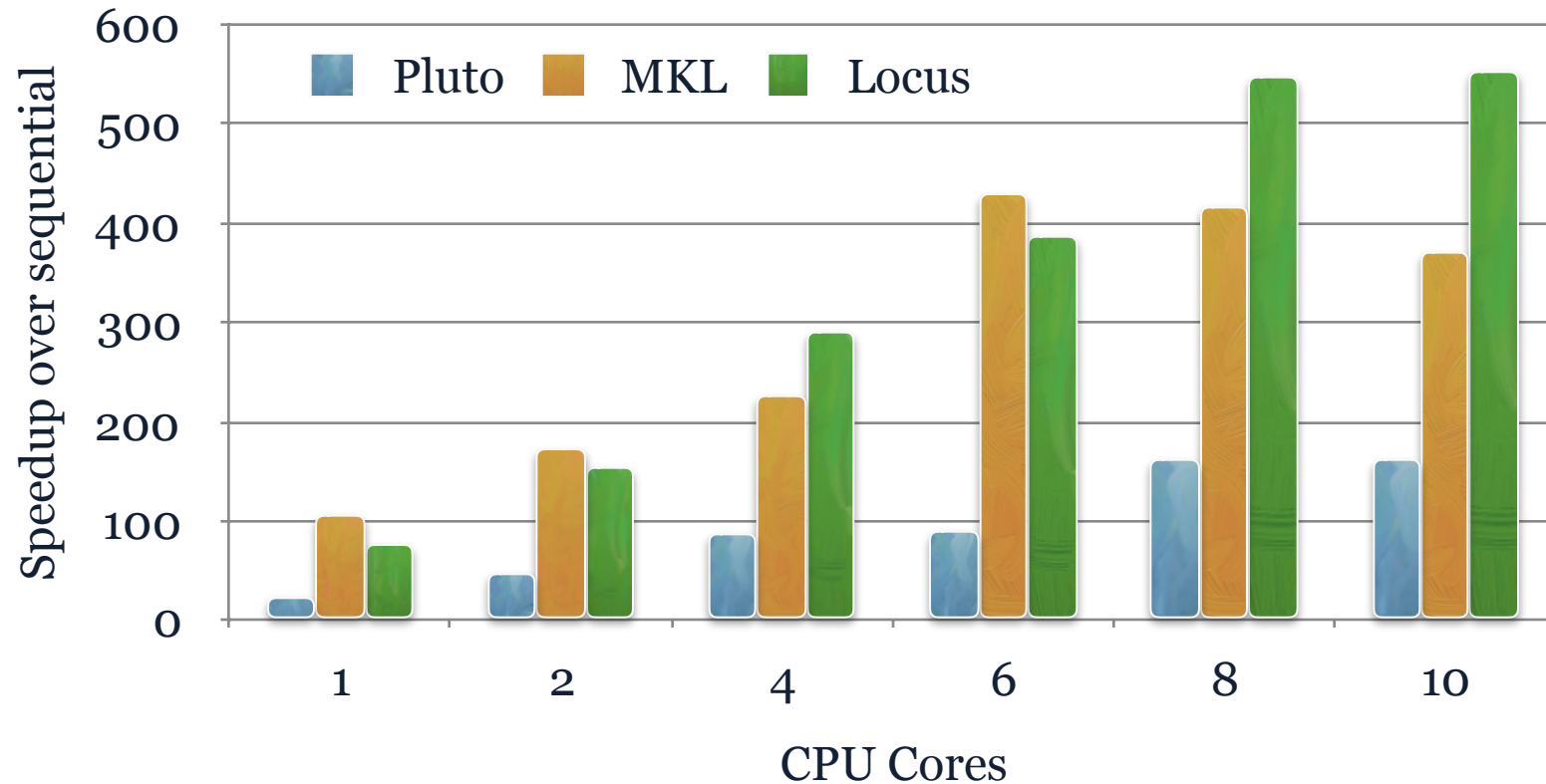


# Experimental Results

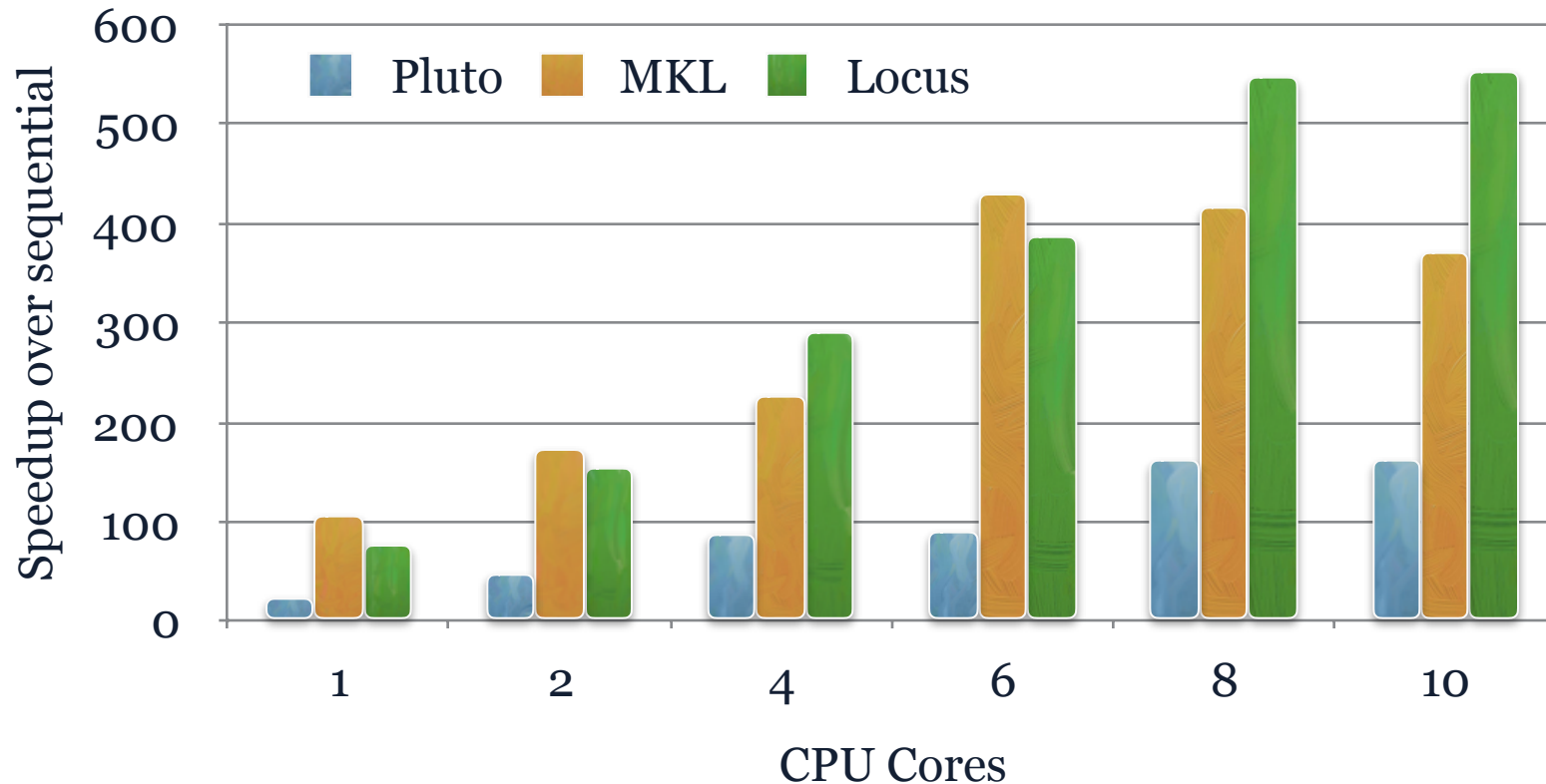
- Intel Xeon E5-2660 10-Core 2.60 GHz
- Compared to Pluto and Intel MKL
  - Default values for parameters, no search
- Examples:
  - Matrix-Matrix Multiplication
  - Stencil Kernels
  - Kripke
  - Arbitrary Loop Nests
- Generic enough to be applied on known and unknown code applications



# Matrix-Matrix Multiplication



# Matrix-Matrix Multiplication



- Empirical search could find very efficient variants
- Comparable with Intel MKL performance



# Matrix-Matrix Multiplication



# Matrix-Matrix Multiplication

Interchange

# Matrix-Matrix Multiplication

Interchange



Tiling

# Matrix-Matrix Multiplication

Interchange



Tiling



Tiling

# Matrix-Matrix Multiplication

Interchange



Tiling

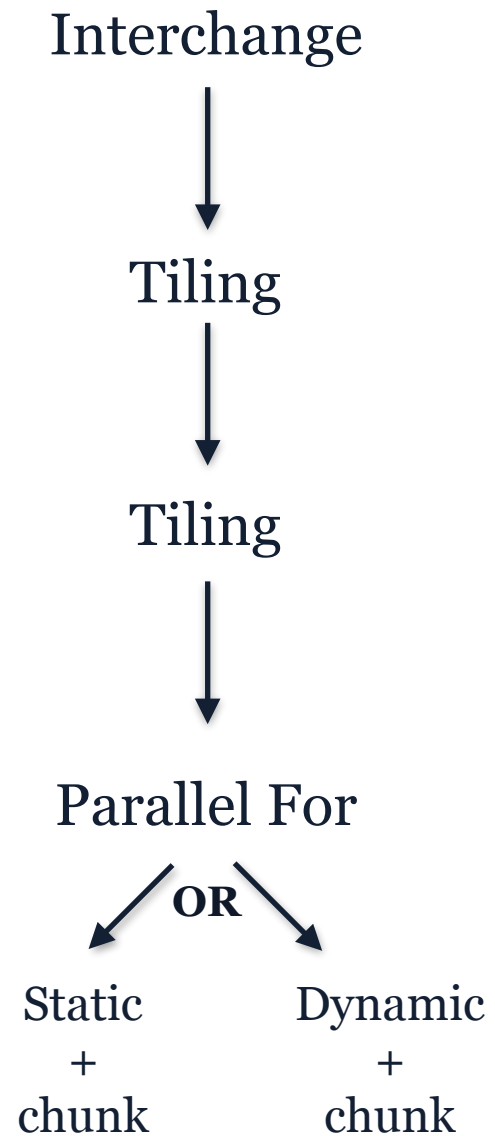


Tiling



Parallel For

# Matrix-Matrix Multiplication



# Matrix-Matrix Multiplication

- Large space of optimization

Interchange



Tiling



Tiling



Parallel For



Static  
+  
chunk

Dynamic  
+  
chunk

# Matrix-Matrix Multiplication

- Large space of optimization
- 34,012,224 possible variants

Interchange



Tiling



Tiling



Parallel For



Static  
+  
chunk

Dynamic  
+  
chunk



# Matrix-Matrix Multiplication

- Large space of optimization
- 34,012,224 possible variants
- Average of ~450 variants evaluated per setup

Interchange



Tiling



Tiling



Parallel For



Static  
+  
chunk

Dynamic  
+  
chunk

# Matrix-Matrix Multiplication

- Large space of optimization
- 34,012,224 possible variants
- Average of ~450 variants evaluated per setup
- 80 minutes search per setup

Interchange



Tiling



Tiling



Parallel For



Static  
+  
chunk

Dynamic  
+  
chunk

# Stencils



# Stencils

- 6 different stencils

# Stencils

- 6 different stencils
- Skew tiling accross time-space

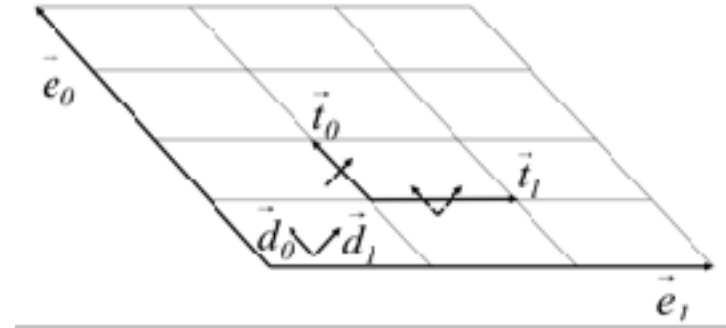
# Stencils

- 6 different stencils
- Skew tiling accross time-space
- Found better tiling shapes



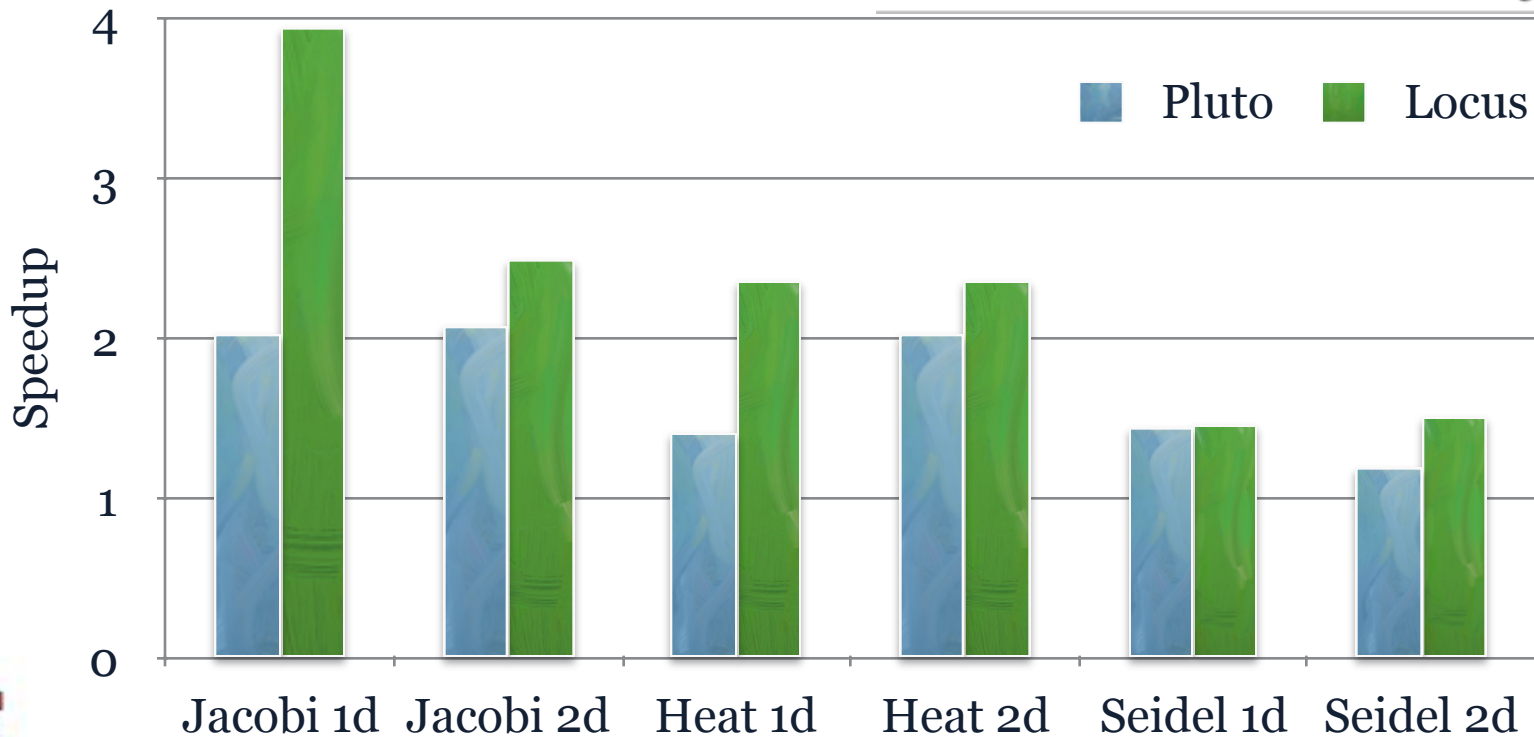
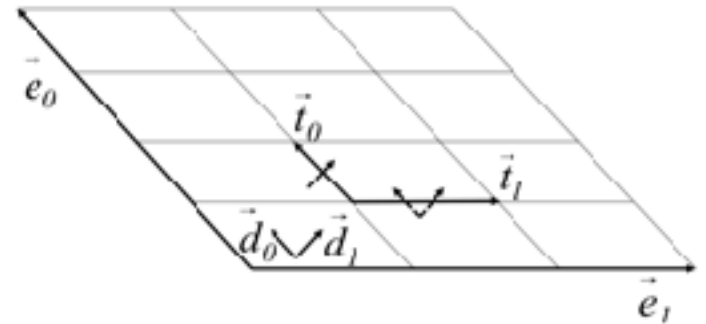
# Stencils

- 6 different stencils
- Skew tiling accross time-space
- Found better tiling shapes



# Stencils

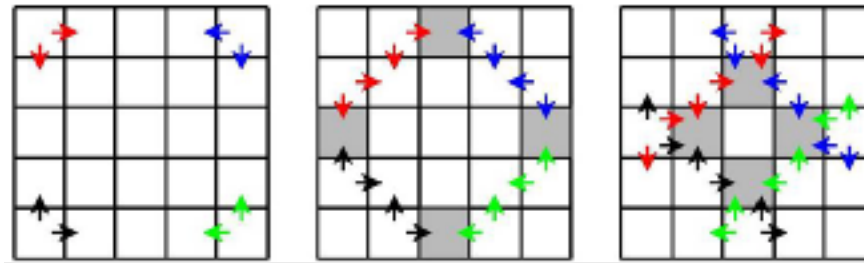
- 6 different stencils
- Skew tiling accross time-space
- Found better tiling shapes



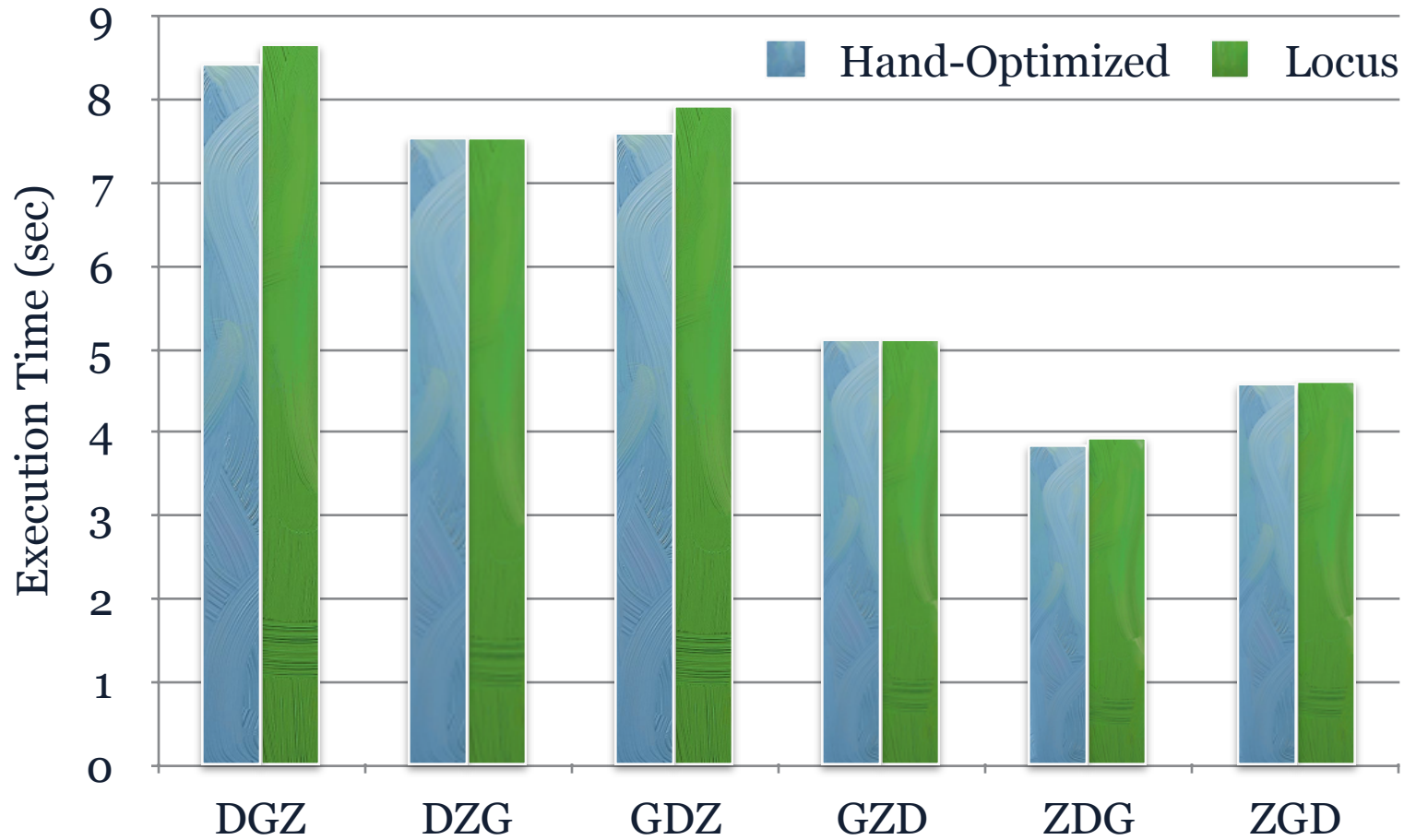


# Kripke

- Deterministic particle transport code and proxy-app for the Ardra project developed at LLNL
- 5 kernels: LTimes, LPlusTimes, Scattering , Source, and Sweep
- 6 hand-optimized versions (6 angular fluxes using a 3D array indexed by direction D, group G and zone Z)
- From a single source code generate the 6 hand-optimized versions using Locus



# Kripke



# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be included here.
          #####

          *phi_out += *sigs * *phi * fraction;
        }
```

# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be in
          #####

          *phi_out += *sigs * *phi * frac
        }
}
```

```
datalayout=enum("DZG","DGZ","GDZ","GZD","ZDG","ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=ompleop);
}
```

# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be in
          #####

          *phi_out += *sigs * *phi * frac
        }
}
```

```
datalayout=enum("DZG", "DGZ", "GDZ", "GZD", "ZDG", "ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=omplloop);
}
```

# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be in
          #####

          *phi_out += *sigs * *phi * frac
        }
    }
```

```
datalayout=enum("DZG","DGZ","GDZ","GZD","ZDG","ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=omplloop);
}
```

# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be inc
          #####

          *phi_out += *sigs * *phi * frac
        }
}
```

```
datalayout=enum("DZG","DGZ","GDZ","GZD","ZDG","ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=omplloop);
}
```

# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

          #####
          # Address calculation to be in
          #####

          *phi_out += *sigs * *phi * frac
        }
}
```

```
datalayout=enum("DZG", "DGZ", "GDZ", "GZD", "ZDG", "ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=ompleop);
}
```



# Kripke - Scattering Kernel

```
for(int nm = 0; nm < num_moments; ++nm)
  for(int g = 0; g < num_groups; ++g)
    for(int gp = 0; gp < num_groups; ++gp)
      for(int zone = 0; zone < num_zones; ++zone)
        for(int mix = z_mixed[z]; mix < z_mixed[z]+num_mixed[z]; ++mix) {
          int material = mixed_material[mix];
          double fraction = mixed_fraction[mix];
          int n = moment_to_coeff[nm];

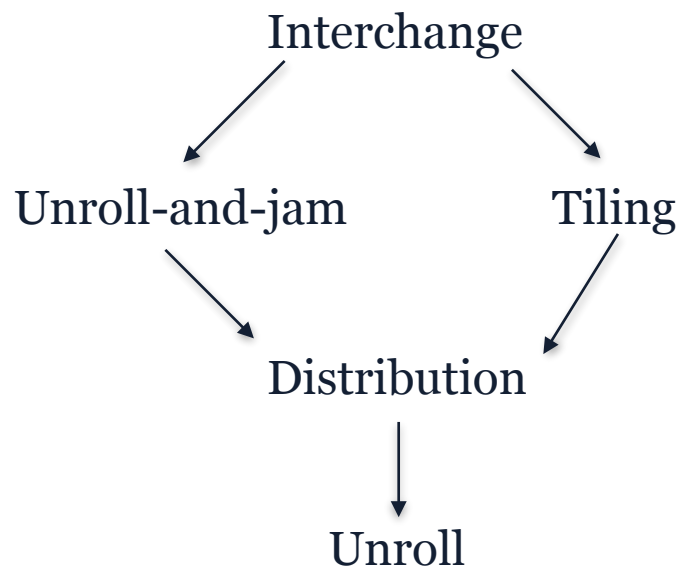
          #####
          # Address calculation to be in
          #####

          *phi_out += *sigs * *phi * frac
        }
    }
```

```
datalayout=enum("DZG", "DGZ", "GDZ", "GZD", "ZDG", "ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
    omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
    looporder=[1,2,0,3,4];
    omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
    looporder=[1,2,3,4,0];
    omploop="0.0.0";
  } elif (datalayout == "ZGD") {
    looporder=[3,4,1,2,0];
    omploop="0";
  } elif (datalayout == "ZDG") {
    looporder=[3,4,0,1,2];
    omploop="0";
  } elif (datalayout == "DZG") {
    looporder=[0,3,4,1,2];
    omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.AltDesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=omplloop);
}
```

# Optimization of Arbitrary Loop Nests

- Generic Locus program to optimize source codes unknown beforehand
- Goal: reproduce Gong Zhangxiaowen et al.<sup>1</sup> work using Locus
- Selected 856 loops from 16 benchmarks
- Transformed loops with all subsets of two sequences:



Benchmark	# of loop nests	Variants assessed
ALPBench [23]	13	39
ASC Sequoia [24]	1	3
Cortexsuite [25]	47	1,297
FreeBench [26]	30	431
Parallel Research Kernels [27]	37	1,055
Livermore Loops [28]	11	121
MediaBench [29]	39	159
Netlib [30]	18	260
NAS Parallel Benchmarks [31]	208	23,384
Polybench [32]	93	7,582
Scimark2 [33]	4	83
SPEC2000 [34]	71	2,228
SPEC2006 [35]	50	216
Extended TSVC [36]	156	6,943
Libraries [37]–[40]	61	1,966
Neural Network Kernels [41]	17	132
Total	856	45,899

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```



Information about  
the code:

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```



Information about  
the code:

- Perfect loop nest?

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```



Information about  
the code:

- Perfect loop nest?
- Loop nest depth

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```



Information about  
the code:

- Perfect loop nest?
- Loop nest depth
- Dependence test available?



# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
    perfect = BuiltIn.IsPerfectLoopNest();
    depth = BuiltIn.LoopNestDepth();
    if (RoseLocus.IsDepAvailable()) {
        if (perfect && depth > 1) {
            permorder = permutation(seq(0,depth));
            RoseLocus.Interchange(order=permorder);
        }
    }
    if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
    }
    } OR {
        if (depth > 1) {
            indexUAJ = integer(1..depth-1);
            UAJfac = poweroftwo(2..4);
            RoseLocus.UnrollAndJam(loop=indexUAJ,
                                factor=UAJfac);
        }
    } OR {
        None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
}
innerloops = BuiltIn.ListInnerLoops();
RoseLocus.Unroll(loop=innerloops,
                 factor=poweroftwo(2..8));
}
```

The image shows a large, dense block of code, likely a Fortran or C-like implementation of the optimization logic shown in the left panel. It contains many conditional statements, loops, and function calls, representing a low-level or compiler-level translation of the high-level logic. The code is organized into several sections, with some parts enclosed in large blocks of comments or preprocessor directives. The overall structure is complex and detailed, reflecting the intricate nature of compiler optimization algorithms.



# I

## 37 lines of code

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define int long long
4  #define endl '\n'
5  #define pi 3.141592653589793
6  #define mod 1000000007
7  #define INF 1e18
8  #define N 1000000
9  #define M 1000000
10 #define S 1000000
11 #define T 1000000
12 #define U 1000000
13 #define V 1000000
14 #define W 1000000
15 #define X 1000000
16 #define Y 1000000
17 #define Z 1000000
18 #define A 1000000
19 #define B 1000000
20 #define C 1000000
21 #define D 1000000
22 #define E 1000000
23 #define F 1000000
24 #define G 1000000
25 #define H 1000000
26 #define I 1000000
27 #define J 1000000
28 #define K 1000000
29 #define L 1000000
30 #define M 1000000
31 #define N 1000000
32 #define O 1000000
33 #define P 1000000
34 #define Q 1000000
35 #define R 1000000
36 #define S 1000000
37 #define T 1000000
38 #define U 1000000
39 #define V 1000000
40 #define W 1000000
41 #define X 1000000
42 #define Y 1000000
43 #define Z 1000000
44 #define A 1000000
45 #define B 1000000
46 #define C 1000000
47 #define D 1000000
48 #define E 1000000
49 #define F 1000000
50 #define G 1000000
51 #define H 1000000
52 #define I 1000000
53 #define J 1000000
54 #define K 1000000
55 #define L 1000000
56 #define M 1000000
57 #define N 1000000
58 #define O 1000000
59 #define P 1000000
60 #define Q 1000000
61 #define R 1000000
62 #define S 1000000
63 #define T 1000000
64 #define U 1000000
65 #define V 1000000
66 #define W 1000000
67 #define X 1000000
68 #define Y 1000000
69 #define Z 1000000
70 #define A 1000000
71 #define B 1000000
72 #define C 1000000
73 #define D 1000000
74 #define E 1000000
75 #define F 1000000
76 #define G 1000000
77 #define H 1000000
78 #define I 1000000
79 #define J 1000000
80 #define K 1000000
81 #define L 1000000
82 #define M 1000000
83 #define N 1000000
84 #define O 1000000
85 #define P 1000000
86 #define Q 1000000
87 #define R 1000000
88 #define S 1000000
89 #define T 1000000
90 #define U 1000000
91 #define V 1000000
92 #define W 1000000
93 #define X 1000000
94 #define Y 1000000
95 #define Z 1000000
96 #define A 1000000
97 #define B 1000000
98 #define C 1000000
99 #define D 1000000
100 #define E 1000000
101 #define F 1000000
102 #define G 1000000
103 #define H 1000000
104 #define I 1000000
105 #define J 1000000
106 #define K 1000000
107 #define L 1000000
108 #define M 1000000
109 #define N 1000000
110 #define O 1000000
111 #define P 1000000
112 #define Q 1000000
113 #define R 1000000
114 #define S 1000000
115 #define T 1000000
116 #define U 1000000
117 #define V 1000000
118 #define W 1000000
119 #define X 1000000
120 #define Y 1000000
121 #define Z 1000000
122 #define A 1000000
123 #define B 1000000
124 #define C 1000000
125 #define D 1000000
126 #define E 1000000
127 #define F 1000000
128 #define G 1000000
129 #define H 1000000
130 #define I 1000000
131 #define J 1000000
132 #define K 1000000
133 #define L 1000000
134 #define M 1000000
135 #define N 1000000
136 #define O 1000000
137 #define P 1000000
138 #define Q 1000000
139 #define R 1000000
140 #define S 1000000
141 #define T 1000000
142 #define U 1000000
143 #define V 1000000
144 #define W 1000000
145 #define X 1000000
146 #define Y 1000000
147 #define Z 1000000
148 #define A 1000000
149 #define B 1000000
150 #define C 1000000
151 #define D 1000000
152 #define E 1000000
153 #define F 1000000
154 #define G 1000000
155 #define H 1000000
156 #define I 1000000
157 #define J 1000000
158 #define K 1000000
159 #define L 1000000
160 #define M 1000000
161 #define N 1000000
162 #define O 1000000
163 #define P 1000000
164 #define Q 1000000
165 #define R 1000000
166 #define S 1000000
167 #define T 1000000
168 #define U 1000000
169 #define V 1000000
170 #define W 1000000
171 #define X 1000000
172 #define Y 1000000
173 #define Z 1000000
174 #define A 1000000
175 #define B 1000000
176 #define C 1000000
177 #define D 1000000
178 #define E 1000000
179 #define F 1000000
180 #define G 1000000
181 #define H 1000000
182 #define I 1000000
183 #define J 1000000
184 #define K 1000000
185 #define L 1000000
186 #define M 1000000
187 #define N 1000000
188 #define O 1000000
189 #define P 1000000
190 #define Q 1000000
191 #define R 1000000
192 #define S 1000000
193 #define T 1000000
194 #define U 1000000
195 #define V 1000000
196 #define W 1000000
197 #define X 1000000
198 #define Y 1000000
199 #define Z 1000000
200 #define A 1000000
201 #define B 1000000
202 #define C 1000000
203 #define D 1000000
204 #define E 1000000
205 #define F 1000000
206 #define G 1000000
207 #define H 1000000
208 #define I 1000000
209 #define J 1000000
210 #define K 1000000
211 #define L 1000000
212 #define M 1000000
213 #define N 1000000
214 #define O 1000000
215 #define P 1000000
216 #define Q 1000000
217 #define R 1000000
218 #define S 1000000
219 #define T 1000000
220 #define U 1000000
221 #define V 1000000
222 #define W 1000000
223 #define X 1000000
224 #define Y 1000000
225 #define Z 1000000
226 #define A 1000000
227 #define B 1000000
228 #define C 1000000
229 #define D 1000000
230 #define E 1000000
231 #define F 1000000
232 #define G 1000000
233 #define H 1000000
234 #define I 1000000
235 #define J 1000000
236 #define K 1000000
237 #define L 1000000
238 #define M 1000000
239 #define N 1000000
240 #define O 1000000
241 #define P 1000000
242 #define Q 1000000
243 #define R 1000000
244 #define S 1000000
245 #define T 1000000
246 #define U 1000000
247 #define V 1000000
248 #define W 1000000
249 #define X 1000000
250 #define Y 1000000
251 #define Z 1000000
252 #define A 1000000
253 #define B 1000000
254 #define C 1000000
255 #define D 1000000
256 #define E 1000000
257 #define F 1000000
258 #define G 1000000
259 #define H 1000000
260 #define I 1000000
261 #define J 1000000
262 #define K 1000000
263 #define L 1000000
264 #define M 1000000
265 #define N 1000000
266 #define O 1000000
267 #define P 1000000
268 #define Q 1000000
269 #define R 1000000
270 #define S 1000000
271 #define T 1000000
272 #define U 1000000
273 #define V 1000000
274 #define W 1000000
275 #define X 1000000
276 #define Y 1000000
277 #define Z 1000000
278 #define A 1000000
279 #define B 1000000
280 #define C 1000000

```

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
  }
  if (perfect) {
    indexT1 = integer(1..depth);
    T1fac = poweroftwo(2..32);
    RoseLocus.Tiling(loop=indexT1, factor=T1fac);
  }
  } OR {
    if (depth > 1) {
      indexUAJ = integer(1..depth-1);
      UAJfac = poweroftwo(2..4);
      RoseLocus.UnrollAndJam(loop=indexUAJ,
                           factor=UAJfac);
    }
  } OR {
    None; # No tiling, interchange, or unroll and jam.
  }
  innerloops = BuiltIn.ListInnerLoops();
  *RoseLocus.Distribute(loop=innerloops);
}
innerloops = BuiltIn.ListInnerLoops();
RoseLocus.Unroll(loop=innerloops,
                 factor=poweroftwo(2..8));
}
```

**37 lines of code**

**1200+ lines of code**

# Optimization of Arbitrary Loop Nests

```
CodeReg scop {  
    perfect = BuiltIn.IsPerfectLoopNest();  
    depth = BuiltIn.LoopNestDepth();  
    if (RoseLocus.IsDepAvailable()) {  
        if (perfect && depth > 1) {  
            permorder = permutation(seq(0,depth));  
            RoseLocus.Interchange(order=permorder);  
        }  
        if (perfect) {  
            indexT1 = integer(1..depth);  
            T1fac = poweroftwo(2..32);  
            RoseLocus.Tiling(loop=indexT1, factor=T1fac);  
        }  
    } OR {  
        if (depth > 1) {  
            indexUAJ = integer(1..depth-1);  
            UAJfac = poweroftwo(2..4);  
            RoseLocus.UnrollAndJam(loop=indexUAJ,  
                                   factor=UAJfac);  
        }  
    } OR {  
        None; # No tiling, interchange, or unroll and jam.  
    }  
    innerloops = BuiltIn.ListInnerLoops();  
    *RoseLocus.Distribute(loop=innerloops);  
}  
innerloops = BuiltIn.ListInnerLoops();
```

**37 lines of code**

- Reproduced Gong Zhangxiaowen et al. results
- Much more concise and flexible



**1200+ lines of code**

**37 lines of code**

# Conclusions

- Locus is able to represent *complex* optimization spaces for different code regions
- Easy to use fine-grain *optimizations* in fine-grain *regions of code* to improve performance
- *Share* resulting optimization programs to amortize the search time
- Keep the baseline version *cleaner* and *simpler* for the long term
- Future work:
  - Use multiple search modules concurrently to speed up the search process
  - Help users at designing optimization sequences



# Acknowledgments

Project is part of the Center for Exascale Simulation of  
Plasma-Coupled Combustion (XPACC)  
[xpacc.illinois.edu](http://xpacc.illinois.edu)

This material is based in part upon work supported by the  
Department of Energy, National Nuclear Security  
Administration, under Award Number DE-NA0002374 and by  
the National Science Foundation under Award 1533912.

We also gratefully acknowledge Gong Zhangxiaowen and  
Justin Szaday for their valuable help in setting up the  
experiments presented for optimizing arbitrary loop nests.



# Locus: A System and a Language for Program Optimization

Thiago Teixeira\*, Corinne Ancourt+, David Padua\*, William Gropp\*  
[tteixei2@illinois.edu](mailto:tteixei2@illinois.edu)

\*Department of Computer Science, University of Illinois at Urbana-Champaign, USA

+MINES ParisTech, PSL University, France



Thank you!