The Polyhedral Model Beyond Loops

Recursion Optimization and Parallelization Through Polyhedral Modeling

Salwa Kobeissi & **Philippe Clauss**

CAMUS team - Inria, University of Strasbourg, ICPS team - ICube Laboratory

Les 13èmes Journées de la Compilation
30 Janvier - 1 Février 2019

Outline

1. Introduction
2. Proposed Solution: From Recursive Functions to Optimized Loops
3. Case Studies
4. Conclusion and Perspectives

Motivation

There may be a huge gap between:

- the statements in a program source code
- the instructions actually performed by a given processor architecture

Motivation

There may be a huge gap between:

- the statements in a program source code
- the instructions actually performed by a given processor architecture

Efficient optimizations may be applied as soon as the actual runtime behavior has been discovered

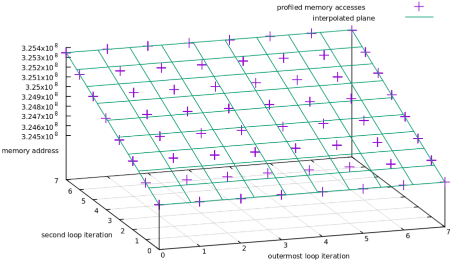
- dedicated to specific control structures & memory access patterns

Inspiration

Apollo

- Captures a polyhedral behavior of loops at runtime
- Applies the polyhedral model

Memory Accesses Behavior at Runtime from statically non-polyhedral loops!



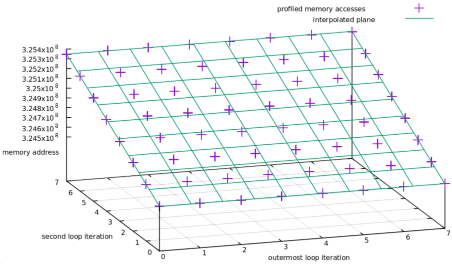
Inspiration

Apollo

- Captures a polyhedral behavior of loops at runtime
- Applies the polyhedral model

We apply the Apollo Approach for codes that are originally not loops!
=> **recursions**

Memory Accesses Behavior at Runtime from statically non-polyhedral loops!



Objectives

We are interested in recursive functions:

1. whose runtime behavior can be modeled as polyhedral loops
2. where the structure of their modeling loops is constant regarding the input

Objectives

We are interested in recursive functions:

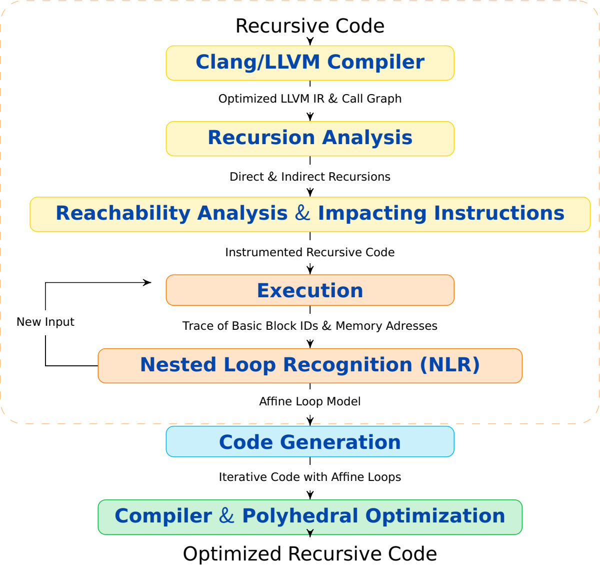
1. whose runtime behavior can be modeled as polyhedral loops
2. where the structure of their modeling loops is constant regarding the input

Objectives

1. optimizing recursive functions through transformation into affine loops
2. extending the scope of polyhedral optimizations to cover recursive functions

1. Introduction
2. Proposed Solution: From Recursive Functions to Optimized Loops
3. Case Studies
4. Conclusion and Perspectives

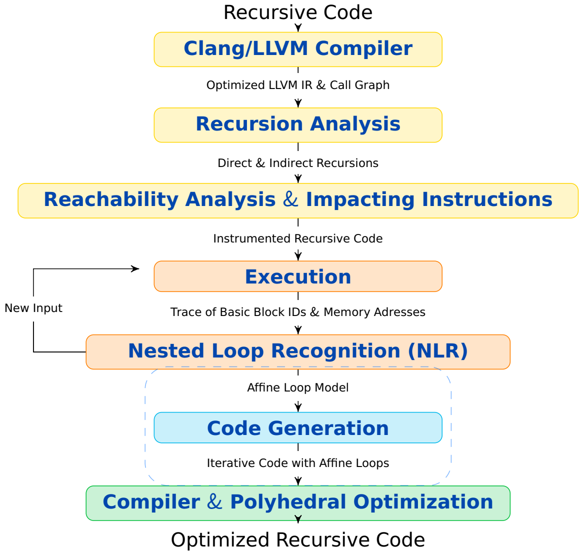
Implementation



The implementation consists of 3 main steps:

1. Recursive Control and Memory Behavior Analysis

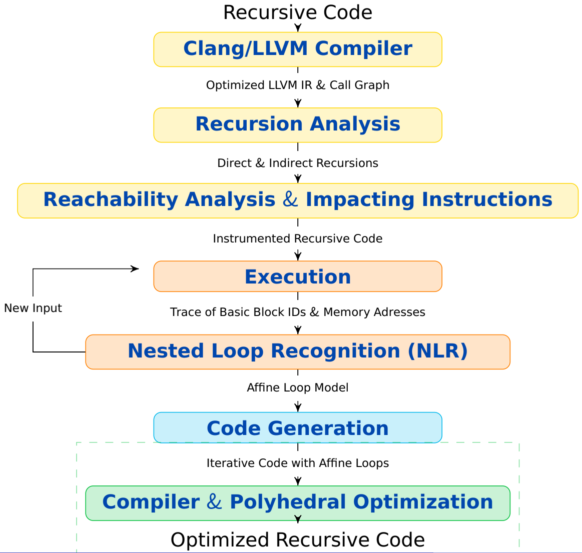
Implementation



The implementation consists of 3 main steps:

1. Recursive Control and Memory Behavior Analysis
2. Recursion to Affine Loop Nest Transformation

Implementation


Recursive Code

Clang/LLVM Compiler

Optimized LLVM IR & Call Graph

Recursion Analysis

Direct & Indirect Recursions

Reachability Analysis & Impacting Instructions

Instrumented Recursive Code

Execution

New Input

Trace of Basic Block IDs & Memory Adresses

Nested Loop Recognition (NLR)

Affine Loop Model

Code Generation

Iterative Code with Affine Loops

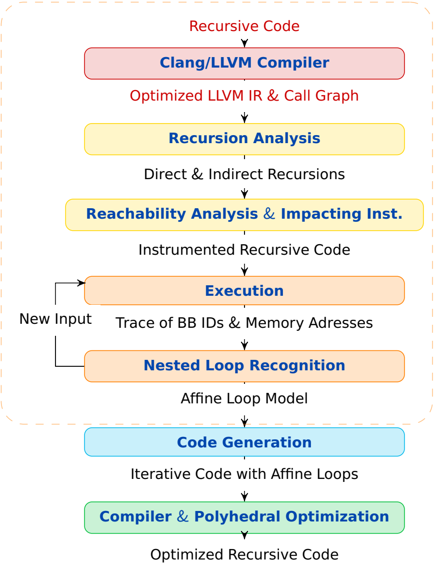
Compiler & Polyhedral Optimization

Optimized Recursive Code

The implementation consists of 3 main steps:

1. Recursive Control and Memory Behavior Analysis
2. Recursion to Affine Loop Nest Transformation
3. Polyhedral Optimizations

Recursive Control and Memory Behavior Analysis



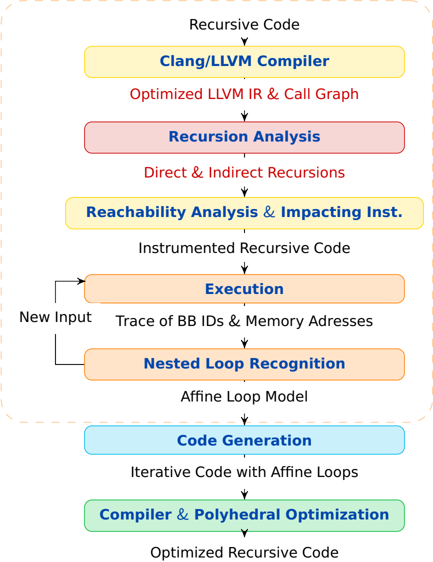
Input: recursive code

Apply classical LLVM optimization passes

- promote memory to register
- simplify CFG
- dead code elimination

Output: optimized LLVM IR & call graph

Recursive Control and Memory Behavior Analysis



Recursive Code

Clang/LLVM Compiler

Optimized LLVM IR & Call Graph

Recursion Analysis

Direct & Indirect Recursions

Reachability Analysis & Impacting Inst.

Instrumented Recursive Code

Execution

New Input

Trace of BB IDs & Memory Adresses

Nested Loop Recognition

Affine Loop Model

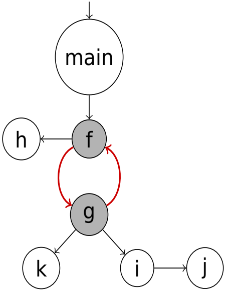
Code Generation

Iterative Code with Affine Loops

Compiler & Polyhedral Optimization

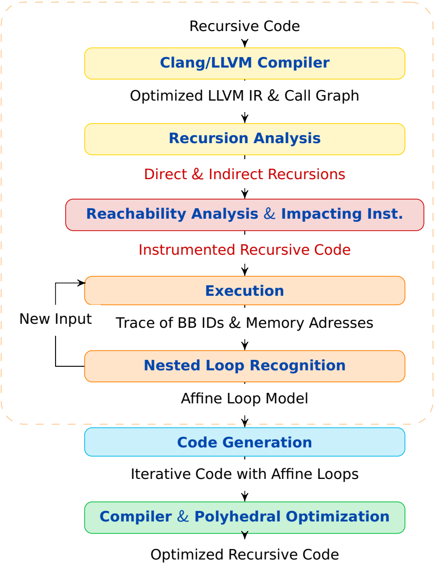
Optimized Recursive Code

Input: optimized IR & call graph



Output: direct & indirect recursions

Recursive Control and Memory Behavior Analysis



Recursive Code

Clang/LLVM Compiler

Optimized LLVM IR & Call Graph

Recursion Analysis

Direct & Indirect Recursions

Reachability Analysis & Impacting Inst.

Instrumented Recursive Code

Execution

New Input

Trace of BB IDs & Memory Adresses

Nested Loop Recognition

Affine Loop Model

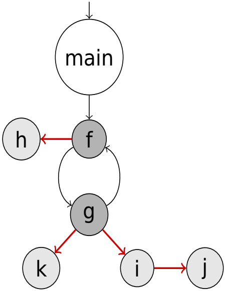
Code Generation

Iterative Code with Affine Loops

Compiler & Polyhedral Optimization

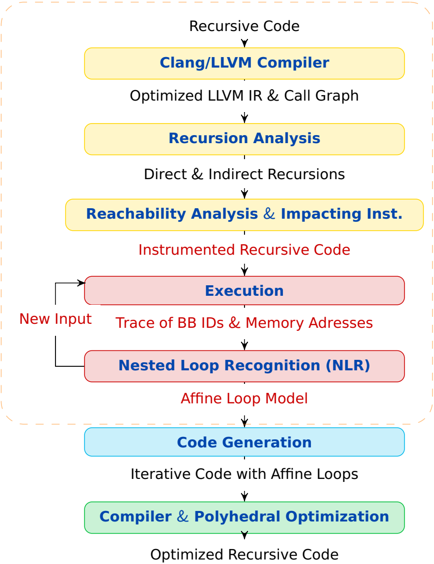
Optimized Recursive Code

Input: direct & indirect recursions



Output: instrumented recursive code

Recursive Control and Memory Behavior Analysis



Input: Trace of the program execution : Basic Block IDs & Memory Addresses

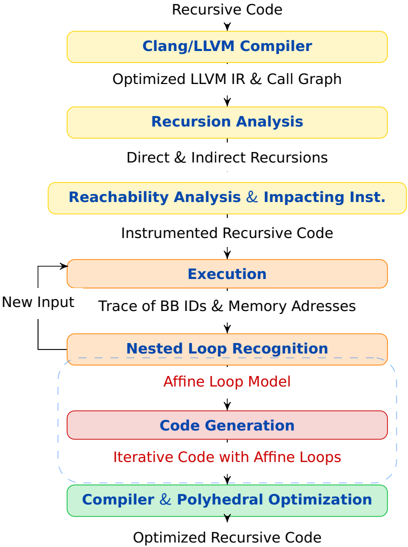
Nested Loop Reconginition (NLR) algorithm applications:

1. program behavior modeling for any measured quantity such as memory accesses
2. execution trace compressing
3. value prediction

(ketterlin & Clauss, GGO 2008)

Output: Affine Loop Model

Recursion to Affine Loop Nest Transformation

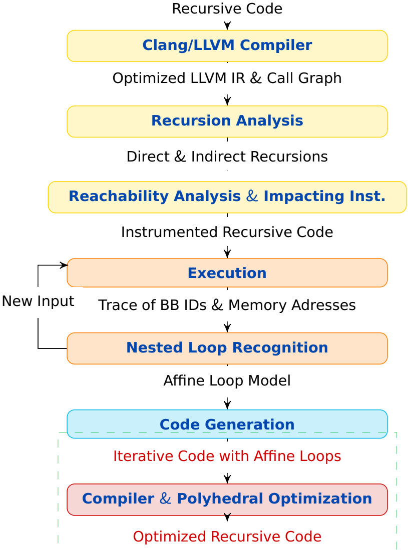


Input: Affine loop model

1. Extract NLR resulting loop nests structures
2. Construct loops in the LLVM IR using:
   - Instrumented basic blocks
   - Interpolated memory addresses

Output: Iterative code with affine loops

Polyhedral Optimizations



Recursive Code

Clang/LLVM Compiler

Optimized LLVM IR & Call Graph

Recursion Analysis

Direct & Indirect Recursions

Reachability Analysis & Impacting Inst.

Instrumented Recursive Code

Execution

New Input

Trace of BB IDs & Memory Adresses

Nested Loop Recognition

Affine Loop Model

Code Generation

Iterative Code with Affine Loops

Compiler & Polyhedral Optimization

Optimized Recursive Code

Input: Iterative code with affine loops

- Apply LLVM optimization passes
- Use polly LLVM polyhedral optimizer (Grosser et al., PPL 2012)

Output: Optimized recursive code

1. Introduction
2. Proposed Solution: From Recursive Functions to Optimized Loops
3. Case Studies
4. Conclusion and Perspectives

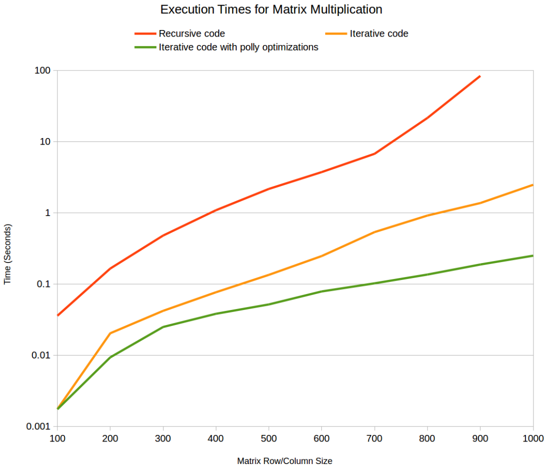
Recursive Matrix Multiplication

```
void MatrixMultiplication(int A[N][N], int B[N][N]){
 static int row=0, column=0, index=0;

   if (row >= N)
          return;

        if(column < N){
          if(index < N){
           C[row][column]+= A[row][index]*B[index][column];
                index++;
                MatrixMultiplication(A, B);
          }
          index=0;
          column++;
          MatrixMultiplication(A, B);
        }
        column=0;
        row++;
        MatrixMultiplication(A, B);
}
```

Recursive Matrix Multiplication Analysis Results

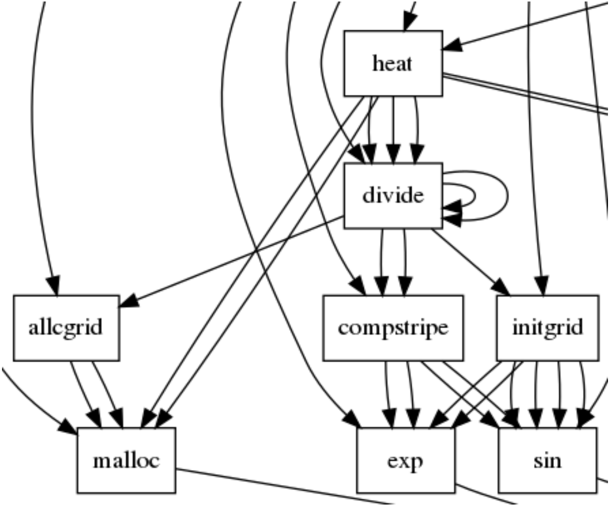
```
for i0 = 0 to N-1
 for i1 = 0 to N-1
  for i2 = 0 to N-1
   val MatrixMultiplication::if.then4 //IR basic block
   ...
   load // memory read
   val MEM1 + 4*N*i0 + 4*i2 //memory address in terms of loops indices
   ... //repetitive memory access patterns
   load
   val MEM2 + 4*i1 + 4*N*i2 //4 is the size of an integer
   ...
   val load
   val MEM3 + 4*N*i0 + 4*i1
   val store // memory write
   val MEM3 + 4*N*i0 + 4*i1
   ...
  val MatrixMultiplication::if.end15
  ...
 val MatrixMultiplication::if.end17
 ...
for i0 = 0 to N*N-1
 for i1 = 0 to N-1
  val MatrixMultiplication::if.end17
  ...
  val MatrixMultiplication::if.end15
  ...
 val MatrixMultiplication::if.end15
 ...
```

Recursive Matrix Multiplication Experimental Results

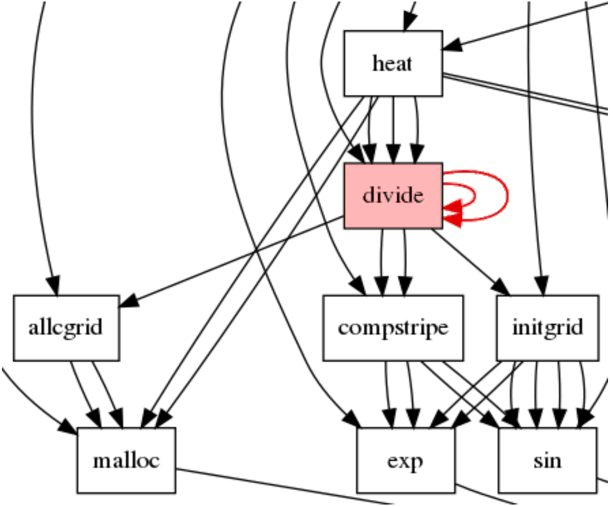


Serial execution (gcc -O3)

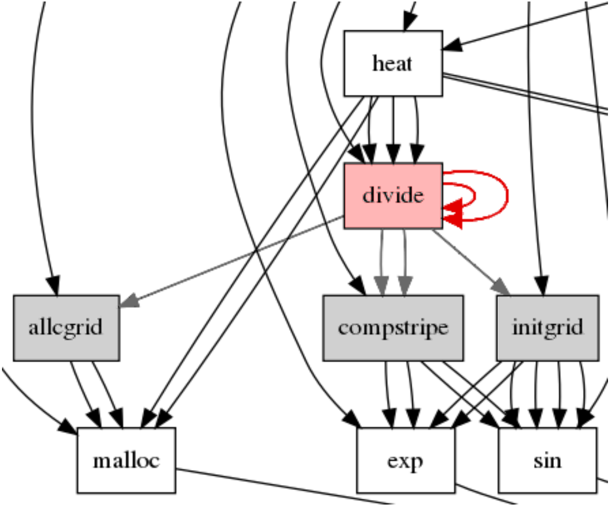
Heat - REAPAR Benchmarks



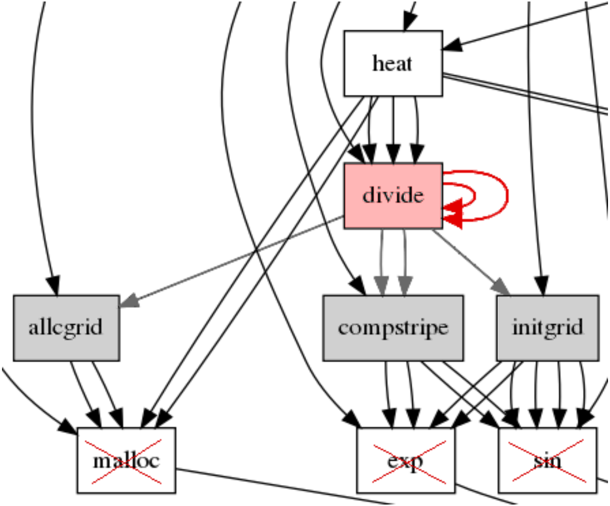
Heat - REAPAR Benchmarks



Heat - REAPAR Benchmarks



Heat - REAPAR Benchmarks



Heat

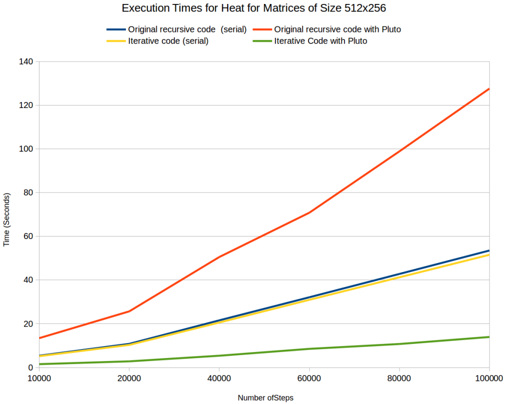
The function compstripe involves interesting linear loops

```
void compstripe(register double **new, register double **old, int lb, int ub)
{
  register int a, b, llb, lub;
  llb = (lb == 0) ? 1 : lb;
  lub = (ub == nx) ? nx - 1 : ub;
  for (a=llb; a < lub; a++) {
    for (b=1; b < ny-1; b++) {
      new[a][b] =  dtdxsq * (old[a+1][b] - 2 * old[a][b] + old[a-1][b])
              + dtdysq * (old[a][b+1] - 2 * old[a][b] + old[a][b-1])
               + old[a][b];
    }
  }
  for (a=llb; a < lub; a++)
    new[a][ny-1] = randb(xu + a * dx, t);
  for (a=llb; a < lub; a++)
    new[a][0] = randa(xu + a * dx, t);
  if (lb == 0) {
    for (b=0; b < ny; b++)
      new[0][b] = randc(yu + b * dy, t);
  }
  if (ub == nx) {
    for (b=0; b < ny; b++)
      new[nx-1][b] = randd(yu + b * dy, t);
  }
}
```

Heat Analysis Results

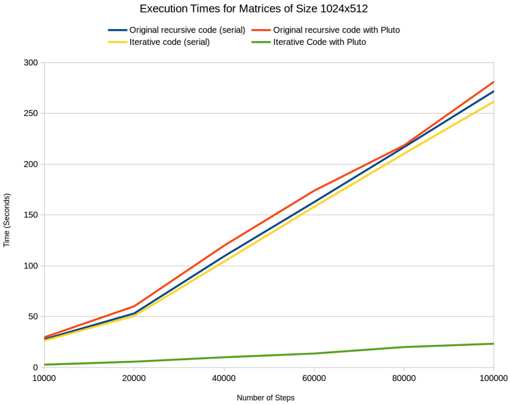
```
for i0 = 0 to Number_of_Steps-1
 for i1 = 0 to 14
  for i2 = 0 to 509
   val compstripe::for.body10, MEM1 + 8224*i1 + 8*i2, MEM2 + 8224*i1 + 8*i2, MEM3 + 8224*i1 + 8*i2
    , MEM4 + 8224*i1 + 8*i2 , MEM5 + 8224*i1 + 8*i2, MEM6 + 8224*i1 + 8*i2
 for i1 = 0 to 14
  val compstripe::for.body63, MEM7 + 8224*i1
 for i1 = 0 to 14
  val compstripe::for.body81, MEM8 + 8224*i1
 for i1 = 0 to 511
  val compstripe::for.body97, MEM9 + 8*i1
 for i1 = 0 to 61
  for i2 = 0 to 15
   for i3 = 0 to 509
    val compstripe::for.body10, MEM10 + 131584*i1 + 8224*i2 + 8*i3, MEM11 + 131584*i1 + 8224*i2 + 8*i3, MEM12 + 131584*i1 + 8224*i2 + 8*i3
     , MEM13 + 131584*i1 + 8224*i2 + 8*i3, MEM14 + 131584*i1 + 8224*i2 + 8*i3, MEM15 + 131584*i1 + 8224*i2 + 8*i3
  for i2 = 0 to 15
   val compstripe::for.body63 , MEM16 + 131584*i1 + 8224*i2
  for i2 = 0 to 15
   val compstripe::for.body81 , MEM17 + 131584*i1 + 8224*i2
 for i1 = 0 to 14
  for i2 = 0 to 509
   val compstripe::for.body10, MEM18 + 8224*i1 + 8*i2, MEM19 + 8224*i1 + 8*i2, MEM20 + 8224*i1 + 8*i2
    , MEM21 + 8224*i1 + 8*i2, MEM22 + 8224*i1 + 8*i2, MEM23 + 8224*i1 + 8*i2
 for i1 = 0 to 14
  val compstripe::for.body63 , MEM24 + 8224*i1
 for i1 = 0 to 14
  val compstripe::for.body81 , MEM25 + 8224*i1
 for i1 = 0 to 511
  val compstripe::for.body115 , MEM26 + 8*i1
   ...........
   ...........
   ...........
   ...........
```

Heat Experimental Results



The codes have been parallelized by Pluto using OpenMP 24 threads (AMD Opteron 6172 2x12-cores - gcc -O3 -fopenmp)

Heat Experimental Results



The codes have been parallelized by Pluto using OpenMP 24 threads (AMD Opteron 6172 2x12-cores - gcc -O3 -fopenmp)

Conclusion

A proof of concept for an automatic recursion-to-affine-loop transformation:

- involving static and dynamic analysis
- transformation passes
- polyhedral optimizers

Achievements

1. extends the polyhedral model applicability to non-loop control structures
2. brings the handled recursive functions to a higher level of optimizations

Future Works

Our future works include:

1. Performing dynamic analysis for recursive behavior at runtime
2. Inducing verification features to obtain a predictive model
3. Tackling input dependent recursive codes

Thank you

Questions ?