COL 729 COMPILER OPTIMIZATION
LAB 3
EXPERIMENTING WITH THE POLYHEDRAL FRAMEWORK

SUBMITTED BY:
NAMRATA JAIN
2018MCS2840

Architecture of polly:

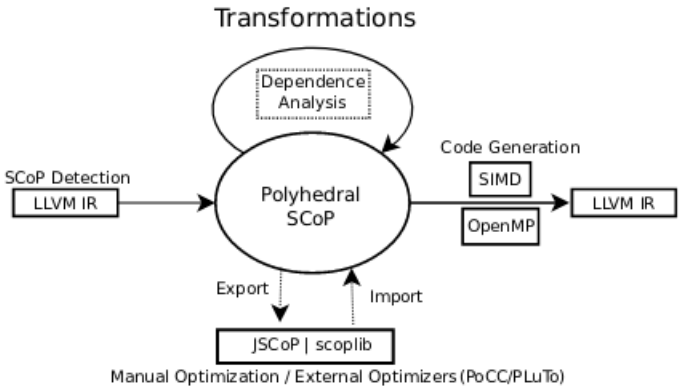


Fig. 1. Architecture of Polly

1. How gemm is optimised using polly?

The matrix multiplication kernel:

```
void matmul(data_type* A, data_type* B, data_type* C) {

for (i=0; i < N; i++)
for (j=0; j < N; j++)
for (k=0; k < N; k++)
C[i][j] += A[k][i] * B[j][k];

}
```

Polly is used to generate optimized vector code.
The first transformation with Polly (+StripMine) changes the loop structure to improve data-locality and to expose a trivially vectorizable loop.

Matrix multiplication kernel with loop structure prepared for vectorization

```
for (k=0; k < N; k++)
```

```
for (j=0; j < N; j+=4)
for (i=0; i < N; i++)
for (jj=j; jj < j + 4; j++)
C[i][jj] += A[k][i] * B[jj][k];
```

Vectorized matrix multiplication kernel

```
for (k=0; k < N; k++)
for (j=0; j < N; j+=4)
for (i=0; i < N; i++)
C[i][j:j+3] += A[k][i] * B[j:3][k];
```

In the next run (+= Vectorization) it takes advantage of the previously created trivially vectorizable loop. Instead of creating the innermost loop, it generates SIMD operations.

Polly automatically generates full vector loads for the access to array C, a stride zero load for the access to A as well as scalar loads for the elements loaded from B, the loads from B is also hoisted out of this loop

By increasing the unrolling limits in the LLVM unrolling pass the inner two loops can be fully unrolled.

Run times

Run times for gemm.c, when compiled with (in seconds)

- **GCC (-O3) : ~5.02**
- **ICC (-O3) : ~3.80**
- **Clang (-O3) without polly extension : ~4.31**
- **Clang (-O3) with polly extension : ~3.81**

2. Analysis of assembly generated

a. GCC (-O3)

- The O3 option turns on optimizations, such as instruction scheduling, function inlining, in addition to all the optimizations of the lower levels -O2 and -O1.

- The assembly generated has less number of SIMD instructions as compared to clang with polly and ICC.

- The optimisations performed :

  -fgcse-after-reload: clean up redundant spilling.

-finline-functions : Consider all functions for inlining, even if they are not declared inline. The compiler heuristically decides which functions are worth integrating in this way.
-fipa-cp-clone : Perform function cloning to make interprocedural constant propagation stronger.
-floop-interchange : improve cache performance on loop nest and allow further loop optimizations, like vectorization, to take place
-floop-unroll-and-jam : Apply unroll and jam transformations on feasible loops. In a loop nest this unrolls the outer loop by some factor and fuses the resulting multiple inner loops.
-fpeel-loops : Peels loops for which there is enough information that they do not roll much

b. ICC (-O3)

- ICC performs loop transformations and introduces more SIMD instructions. Its code is almost twice as fast as the one generated with LLVM. Yet, it still requires a large number of scalar loads,which suggests that further optimization is possible.

- O3 option enables global optimization which includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. This option also disables inlining of some intrinsics. The compiler vectorization is enabled at O2 and higher levels.

- It also enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets.

- ICC uses SIMD instructions like:

  a. stmxcsr : store Streaming SIMD Extension control/status word from m32.

  b. pshufd : Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand)

  equivalent sse instruction : __m128i _mm_shuffle_epi32 (__m128i a, int imm8)

  c. movdqa : For moving data

  equivalent sse instruction : __m128i _mm_load_si128 (__m128i *p) ,_mm_store_si128 (__m128i *p, __m128i a)

  d. movdqu : Move unaligned double quadword from xmm2/m128 to xmm1.

  equivalent sse instruction : __m128i _mm_loadu_si128 (__m128i const* mem_addr)

  e. pmuludq : Multiply the low unsigned 32-bit integers from each packed 64-bit element in a and b, and store the unsigned 64-bit results in dst.

equivalent sse instruction:__m128i _mm_mul_epu32 (__m128i a, __m128i b)

f. paddd : Add packed 32-bit integers in a and b, and store the results in dst.

equivalent sse instruction:__m128i _mm_add_epi32 (__m128i a, __m128i b)

c. Clang (-O3) without polly extension

- Optimisations performed :

i. -loop-reduce: Loop Strength Reduction

ii. -loop-rotate: Rotate Loops
iii. -loop-simplify: Canonicalize natural loops
iv. -loop-unroll: Unroll loops
v. -loop-unroll-and-jam: Unroll and Jam loops
vi. -loop-unswitch: Unswitch loops
vii. -loweratomic: Lower atomic intrinsics to non-atomic form
viii. -lowerinvoke: Lower invokes to calls, for unwindless code generators
ix. -lowerswitch: Lower SwitchInsts to branches
x. -mem2reg: Promote Memory to Register

- Clang (without polly extension) uses less number of SIMD instructions
- SIMD instructions used:

a. mulpd: Multiply packed double-precision (64-bit) floating-point elements and store the results in dst.

b. movupd: Load 128-bits (composed of 2 packed double-precision (64-bit) floating-point elements) from memory into dst. mem_addr does not need to be aligned on any particular boundary.

c. addpd : Add packed double-precision (64-bit) floating-point elements in a and b, and store the results in dst

d. Clang (-O3) with polly extension

Polly uses polyhedral techniques to optimize for data-locality and parallelism. First, it detects the parts of a program(SCoP) that will be optimized and translates them into a polyhedral representation. Then it analyses and optimizes the polyhedral representation. Then optimized program code is generated.

- Static Control Parts (SCoPs) of a function are parts of a program in which all control flow and memory accesses are known at compile time. As a result, they can

be described in detail and a precise analysis is possible. Polly currently focuses on detecting and analysing SCoPs .

- Polly provides an advanced dependency analysis and is the place for polyhedral optimizations. At the moment, Polly itself does not perform any optimizations, but allows the export and reimport of its polyhedral representation. The exported representation can be used to manually perform optimizations.

- The largest performance improvements appear not because of the SIMD code introduced, but because of further optimization opportunities exposed through our preparing loop transformations.

- SIMD instructions used:

  cvtsi2sd : convert packed doubleword integer to floating point and vice versa
  movupd : move unaligned packed double precision floating point
  mulpd: Multiply packed double-precision (64-bit) floating-point elements in a and b, and store the results in dst.

  Mulupd

  addsd

Run time of manually optimised source code

Clang (-O3) without polly extension : ~2.94 s
Clang (-O3) with polly extension : ~1.78 s

3. Strengths of polly framework

- The polly project implements a suite of cache-locality optimizations as well as auto-parallelism and vectorization using a polyhedral model.

- The procedure used by polly :how to extract relevant program parts, how to translate theminto a polyhedral representation, how to apply optimizations and how to generate optimized program code is not bound to a specifichigh-level programming language and does not require the input code to exhibit any syntactic format. As a result, constructs such as GOTO loops or pointer arithmeticcan be optimized.

- Polly detects available parallelism automatically and generates optimized OpenMP or SIMD code. This allows optimizers to focus on the parallelisation problem itself and to offload low-level details to Polly.

- It perform classical loop transformations, especially tiling and loop fusion to improve data-locality.

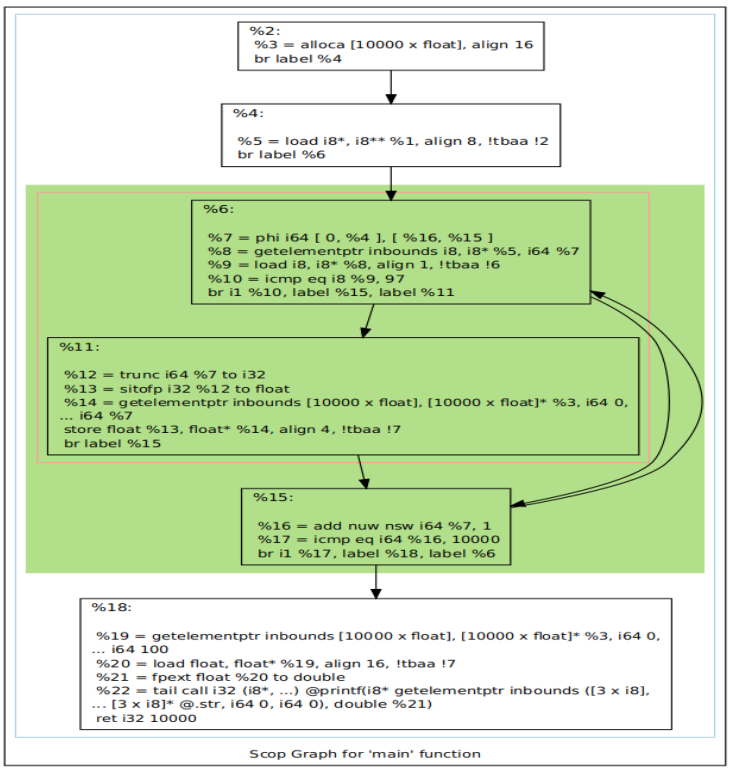
- Polly can apply advanced, polyhedral transformations without the need of any source code annotations or the use of source-to-source techniques. As it works directly inside a compiler, it can be used to optimize existing programs fully automatically.

- As it abstracts away all language specific details and code generation problems, it allows to focus on the high-level optimization problems. With the interface for external optimizers new transformations can be added to Polly without the need to understand all LLVM internals

- Example:

a.

```
#include<stdio.h>

int main(int argc,char **argv)
{
int i;int buckets=10000;
float count[buckets];
char *string=argv[0];
for (i = 0; i < buckets; i++)
{  if(string[i]!='a')
   count[i] = (float)i;
    //printf("%d ",count[i]);
}
printf("%f",count[100]);
return i;
}
```



Polly identifies that the condition inside loop can be modelled as static and thus detects ScoP which can be parallelised.

4. Limitations of polly framework

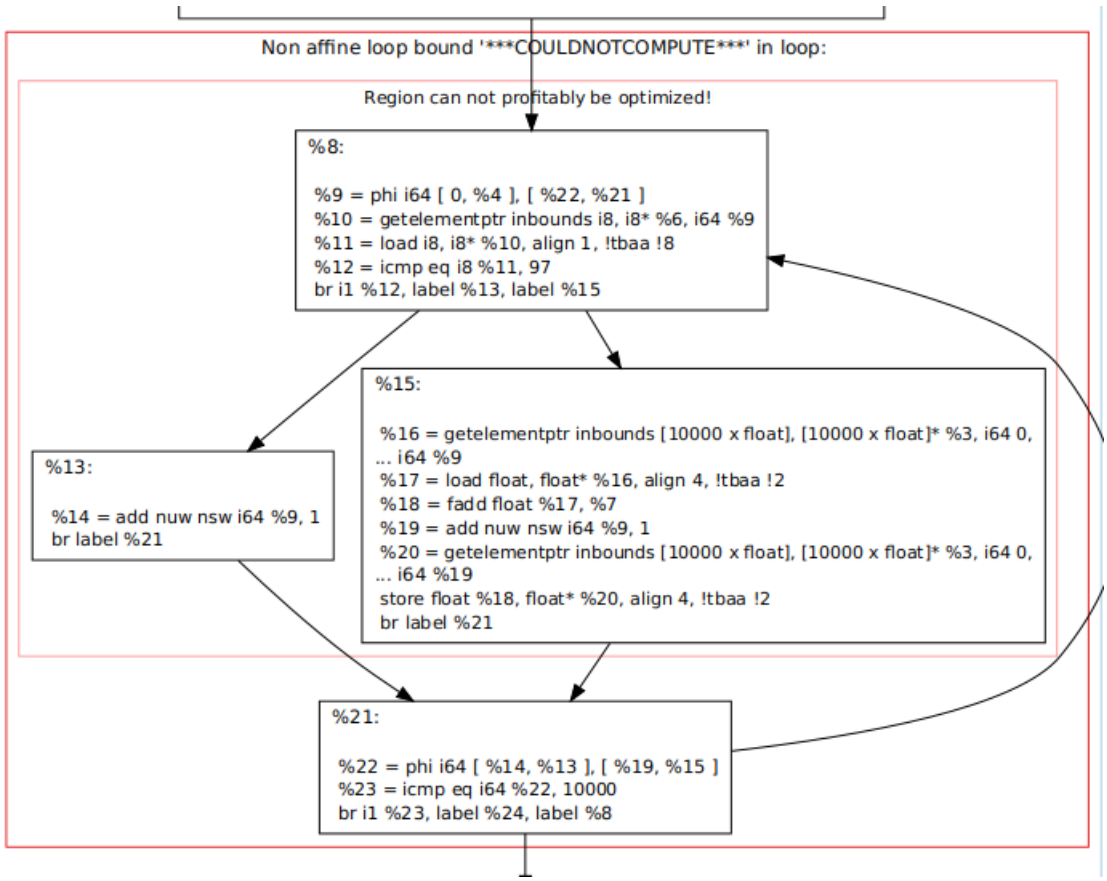
Examples:

Polly could not detect a ScoP for the following :

a. The loop can be parallelised if it can be identified that count[i+1]=1+argc*(i+1)

```c
#include<stdio.h>

int main(int argc,char **argv)
{
   int i;int limit=10000;
   float count[limit];
   count[0]=1;
   char *string=argv[0];
   for (i = 0; i < limit; i++)
   {  if(string[i]!='a')
        count[i+1] = count[i]+(argc) ;
    |
   }
   printf("%f",count[100]);
   return i;
}
```

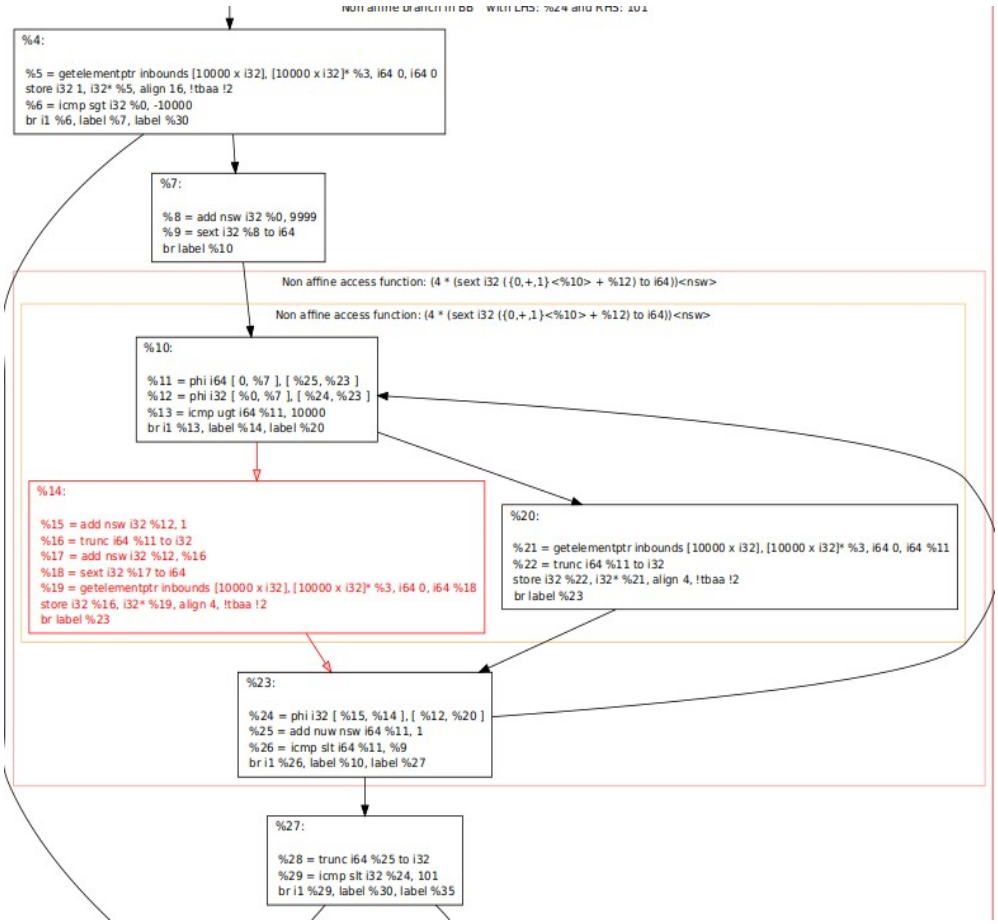
The following CFG generated by polly does not generate Scop because of non affine loop bound.



b. Polly could not do loop fission to separate parallel portion from serial portion of the loop

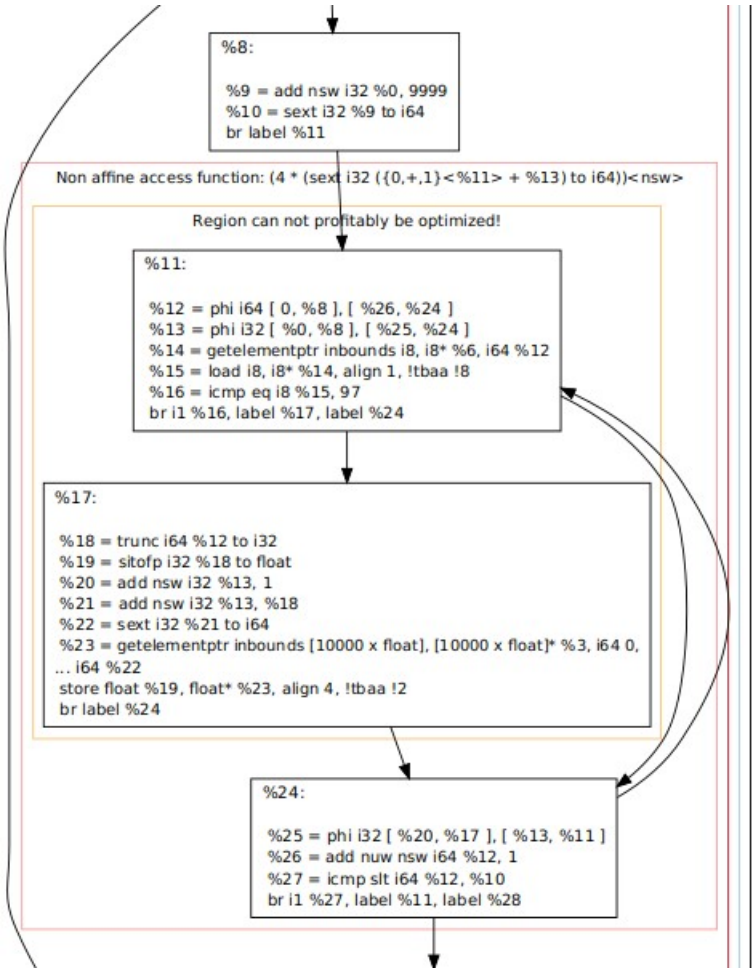
loop can be parallelized for i=0 to i=limit

```c
#include<stdio.h>

int main(int argc,char **argv)
{
   int i;int limit=10000;
   int count[limit];
   count[0]=1;
   int j=argc;
   char* string=argv[0];
   for (i = 0; i < limit+j; i++)
   {
     if(i>limit)
     count[i+argc++]=i ;
     else
     count[i]=i;

   }
   if(argc<=100)
   printf("%d",count[100]);
   return i;
}
```



c. The loop can be parallelized : the index of count is i+argc+k where k can be introduced to keep track if the condition is satisfied or not.

```c
#include<stdio.h>

int main(int argc,char **argv)
{
   int i;int limit=10000;
   float count[limit];
   count[0]=1;
   int j=argc;
   char* string=argv[0];
   for (i = 0; i < limit+j; i++)
   {
     if(string[i]=='a')
     count[i+argc++] = (float)i ;
   }
   if(argc<=100)
   printf("%f",count[100]);
   return i;
}
```



c. Several improvements can be made to increase the number of codes that canbe handled by Polly. Modeling the memory behaviour of certain function calls and intrinsics, especially calls to memcpy, memove and memset.

d. Adding support for cast and modulo operations in the affine expressions is possi-ble since isl supports modulo arithmetic.

e. Dynamically allocated multi-dimensionalarrays are represented as one-dimensional arrays with non-affine subscripts (e.g.,A[n·i+j] instead ofA[i][j] withnbeing the size of the inner dimension)