Graphite: the polyhedral framework of GCC

Sebastian Pop

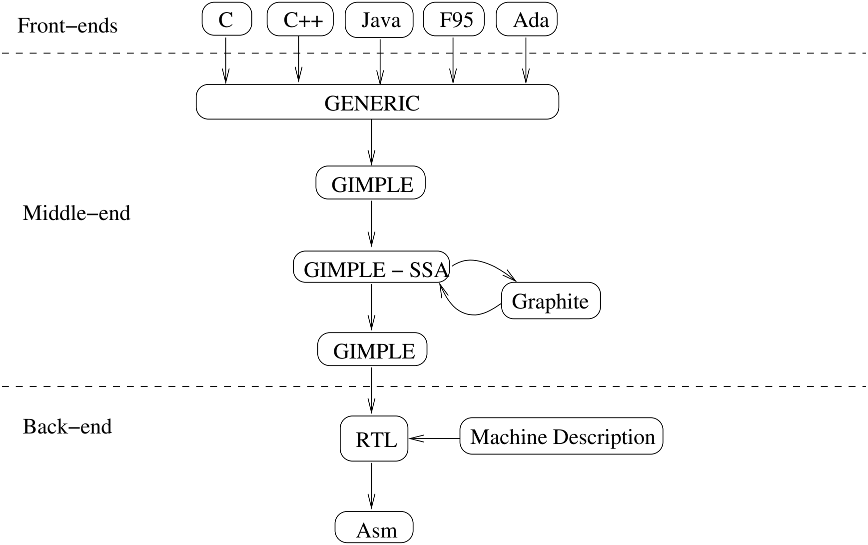
AMD - Austin, Texas

October 20, 2010

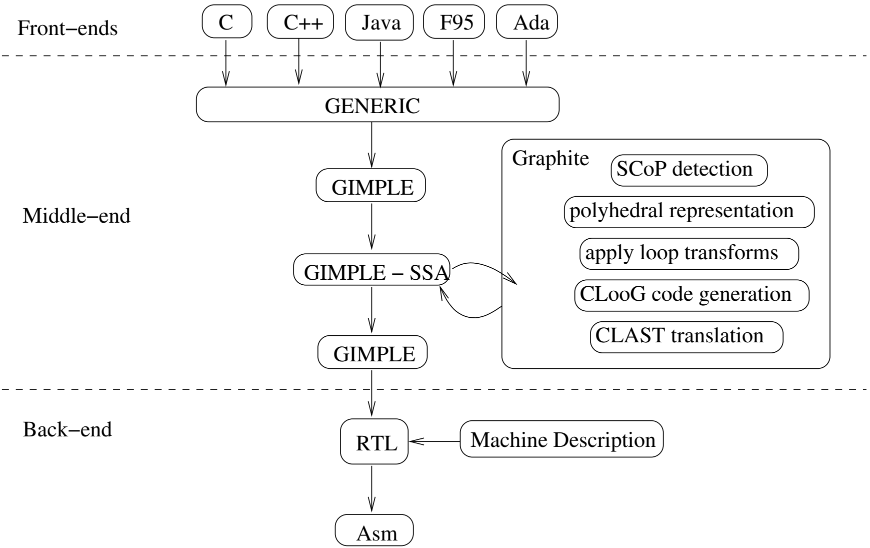
Outline

- Graphite in GCC
- detection of SCoPs
- polyhedral representation
- code generation: CLooG
- loop transforms: blocking, flattening, autopar, autovect

Graphite in GCC



Components of Graphite

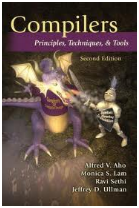


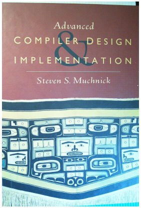
Gimple, SSA, CFG, natural loops

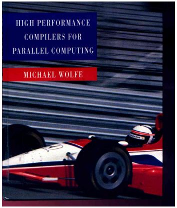
- Gimple = 3 address code
- SSA = Static Single Assignment
- CFG = Control Flow Graph
- Natural loops = strongly connected components of the CFG

Reference books:

- the "Dragon Book" (Aho, Lam, Sethi, Ullman)
- Steven Muchnick
- Michael Wolfe







```
int foo (int, int);
void xxx(void)
{
  int res = 0, x;
  for (x = 45; x > 0; x--)
    res = foo (x, res);
  return res;
}
```

```
--:--  loop-1.c        (C Abbrev)--L6--C0--All----------------------------------------
```

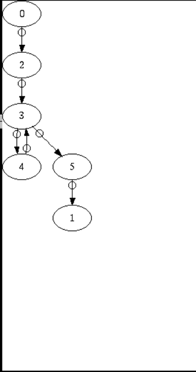
```
$2 = void
(gdb) p debug_loops (3)
loop_0 (header = 0, latch = 1, niter = )
{
  bb_2 (preds = {bb_0 }, succs = {bb_3 })
  {
  <bb 2>:

  }
  bb_5 (preds = {bb_3 }, succs = {bb_1 })
  {
  <bb 5>:
    return;

  }
  loop_1 (header = 3, latch = 4, niter = , upper_bound = 45, estimate = 45)
  {
    bb_3 (preds = {bb_4 bb_2 }, succs = {bb_4 bb_5 })
    {
    <bb 3>:
      # x_11 = PHI <x_7(4), 45(2)>
      # res_10 = PHI <res_5(4), 0(2)>
      res_5 = foo (x_11, res_10);
      x_7 = x_11 + -1;
      if (x_7 > 0)
        goto <bb 4>;
      else
        goto <bb 5>;

    }
    bb_4 (preds = {bb_3 }, succs = {bb_3 })
    {
    <bb 4>:
      goto <bb 3>;

    }
  }
}
$3 = void
(gdb)
```



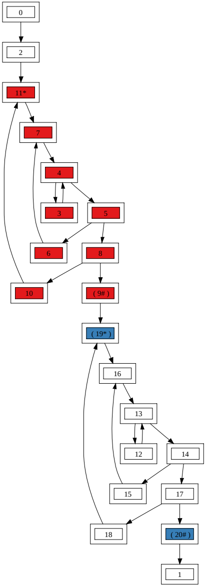

```
--:**  *gud-cc1*       (Debugger:run)--L78--C0--Bot----------------------------------
```

SCoP detection

SCoP: Static Control Part is a region with no side effects

- induction variables (IVs) affine
- canonical IV: one per loop, from 0 to number of iterations, with steps of 1
- linear loop bounds (linear = function of params and outer IVs)
- linear memory accesses
- side effects: function calls, inline asm, volatile, etc.
- regular control flow: irreducible strongly connected components not handled
- basic blocks with no memory accesses not represented
- statements = basic blocks with regular memory accesses

SCoP example



Translation to polyhedral representation

- build Polyhedral Black Boxes (PBB): one statement, a sequence of statements, one basic block, or a single entry single exit (SESE) region.
- record original PBB schedule
- loop nest around PBB
- conditions around PBB
- find SCoP parameters
- record SCoP context: constraints on parameters
- iteration domains: constraints on IV
- data accesses in PBB
- build the data dependence graph

Representation of scalar dependences in Graphite

- the SSA represents dependences between scalars.
- when the scalar dependences cross the boundary of PBBs, we have to expose these dependences to the polyhedral framework: translate scalars into arrays (for a scalar variable "s", define an array of one element and replace all the occurrences of the scalar variable by "S[0]").
- commutative associative reductions are special cased in the data dependence test to remove unwanted dependences.

Polyhedral representation

1. scop context = constraints on parameters
2. iteration domain = bounds of enclosing loops

```
for (i=0; i<m; i++)
 for (j=5; j<n; j++)
  A[2*i][j+1] = ...
```

i	j	m	n	cst	
1	0	0	0	0	$i \geq 0$
-1	0	1	0	-1	$-i + m - 1 \geq 0$
0	1	0	0	5	$j \geq 5$
0	-1	0	1	-1	$-j + n - 1 \geq 0$

Polyhedral representation

1. scop context = constraints on parameters
2. iteration domain = bounds of enclosing loops
3. schedule = execution time (static + dynamic)

- sequence $[\![s_1; s_2]\!]$:
  $\mathcal{S}[\![s_1]\!] = \mathrm{t}, \qquad \mathcal{S}[\![s_2]\!] = \mathrm{t} + 1$
- loop $[\![loop_1\ s\ end_1]\!]$: i_1 indexes $loop_1$ iterations: dynamic time
  $\mathcal{S}[\![loop_1]\!] = \mathrm{t}, \quad \mathcal{S}[\![s]\!] = (\mathrm{t},\ i_1,\ 0)$

Polyhedral representation

1. scop context = constraints on parameters
2. iteration domain = bounds of enclosing loops
3. schedule = execution time (static + dynamic)
4. access functions = data reference accesses

```
for (i=0; i<m; i++)
 for (j=5; j<n; j++)
  A[2*i][j+1] = ...
```

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right] \quad \begin{array}{c} 2*i \\ j+1 \end{array}$$

Polyhedral representation

1. scop context = constraints on parameters
2. iteration domain = bounds of enclosing loops
3. schedule = execution time (static + dynamic)
4. access functions = data reference accesses

data dependences = ILP solution of all these constraints

Data dependence analysis

- data dependences characterize computation sharing
- sharing $\Rightarrow$ synchronization and communications
- no sharing = parallelism = recomputations (privatization)
- legality of a transform = satisfy original computation order

Counting points in polyhedra

In many program analyses and optimizations, questions starting with "how many" need to be answered:

- How many memory locations are touched by a loop?
- How many operations are performed by a loop?
- How many cache lines are touched by a loop?
- How many array elements are accessed between two points?
- How many array elements are live at a given iteration?
- How many times is a statement executed before an iteration?
- How many cache misses does a loop generate?
- How much memory is dynamically allocated?

Techniques used for counting points:

- Ehrhart polynomials
- Barvinok's generating functions

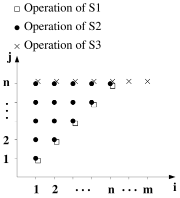
Loop transforms

- Graphite represents the static and dynamic schedules under a polyhedral format: the scattering polyhedra
- identifying statements belonging to a loop, or updating the sequence of statements on the polyhedral representation is difficult
- the LST = Loop Statement Tree represents the statement sequence and loop nesting, but does not include informations about the iteration domains
- loop transformations are performed on the LST and then impacted on the scattering polyhedra

Code generation

- the code generation of an imperative language from the polyhedral representation introduces imperative language constructs: sequence, loops, parallel computations, communication, . . .
- call CLooG for code generation, produces a representation CLAST: CLooG Abstract Syntax Trees
- generate GIMPLE-SSA from CLAST

CLooG's code generation from polyhedra



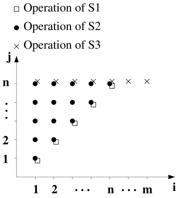
$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

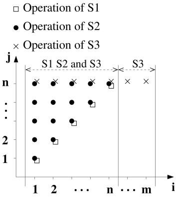
$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq m \\ j = n \end{cases}$$

(a) Initial domains to scan

CLooG's code generation from polyhedra





$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq m \\ j = n \end{cases}$$

(a) Initial domains to scan

```
do i=1, n
```

$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

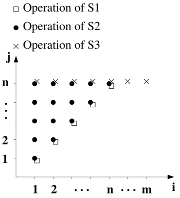
$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq n \\ j = n \end{cases}$$

```
do i=n+1, m
```

$$\mathcal{T}_{S_3} : \begin{cases} n+1 \leq i \leq m \\ j = n \end{cases}$$

(b) Projection and separation onto the first dimension

CLooG's code generation from polyhedra

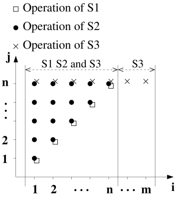


$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq m \\ j = n \end{cases}$$

(a) Initial domains to scan




```
do i=1, n
```

$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

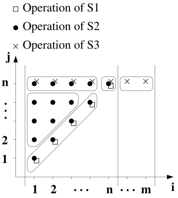
$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq n \\ j = n \end{cases}$$

```
do i=n+1, m
```

$$\mathcal{T}_{S_3} : \begin{cases} n+1 \leq i \leq m \\ j = n \end{cases}$$

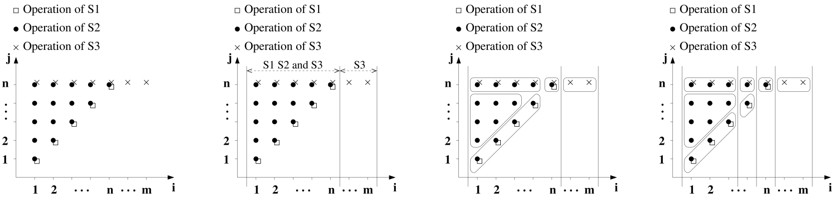
(b) Projection and separation onto the first dimension




```
do i=1, n
  if (i==n) then
    S1(j=n)
    S2(j=n)
    S3(j=n)
  if (i<=n-1) then
    S1(j=i)
    S2(j=i)
  do j=i+1, n-1
    S2
  if (i<=n-1) then
    S2(j=n)
    S3(j=n)
do i=n+1, m
  S3(j=n)
```

(c) Recursion on next dimension

CLooG's code generation from polyhedra



$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq m \\ j = n \end{cases}$$

(a) Initial domains to scan

```
do i=1, n
```

$$\mathcal{T}_{S_1} : \begin{cases} 1 \leq i \leq n \\ j = i \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} 1 \leq i \leq n \\ i \leq j \leq n \end{cases}$$

$$\mathcal{T}_{S_3} : \begin{cases} 1 \leq i \leq n \\ j = n \end{cases}$$

```
do i=n+1, m
```

$$\mathcal{T}_{S_3} : \begin{cases} n+1 \leq i \leq m \\ j = n \end{cases}$$

(b) Projection and separation onto the first dimension

```
do i=1, n
  if (i==n) then
    S1(j=n)
    S2(j=n)
    S3(j=n)
  if (i<=n-1) then
    S1(j=i)
    S2(j=i)
  do j=i+1, n-1
    S2
  if (i<=n-1) then
    S2(j=n)
    S3(j=n)
do i=n+1, m
  S3(j=n)
```

(c) Recursion on next dimension

```
do i=1, n-2
  S1(j=i)
  S2(j=i)
  do j=i+1, n-1
    S2
  S2(j=n)
  S3(j=n)
S1(i=n-1,j=n-1)
S2(i=n-1,j=n-1)
S2(i=n-1,j=n)
S3(i=n-1,j=n)
S1(i=n,j=n)
S2(i=n,j=n)
S3(i=n,j=n)
do i=n+1, m
  S3(j=n)
```

(d) Backtrack with dead code removing

Code generation details

- type of induction variables (IV): when a transform increases the number of iterations, the original IV type may not be large enough to contain all the values of the new IV: use the scop context to get an approximation of the largest integer of the new IV, then compute the smallest type that can represent IV
- to backup original code, use SESE versioning:

```
if (0) {
  original code;
} else {
  transformed code;
}
```

- one could replace the 0 with a runtime condition that validates additional assumptions under which a transform is legal

Examples

- strip mining
- interchange: improves spatial and temporal data locality
- loop blocking (tiling): strip mining + interchange
- loop flattening: removes loops, increases ILP, avoids bubbles in processors' pipeline

Loop blocking

- original loop nest:

```
for (i = 0; i < 1000; i++)
  for (j = 0; j < 1000; j++)
    a[i][j] = b[i][j] + 1;
```

- strip mining (with strides of 64):

```
for (s1 = 0; s1 <= 15; s1++)
  for (i = 64*s1; i <= min (64*s1 + 63, 999); i++)
    for (s3 = 0; s3 <= 15; s3++)
      for (j = 64*s3; j <= min (64*s3 + 63, 999); j++)
        a[i][j] = b[i][j] + 1;
```

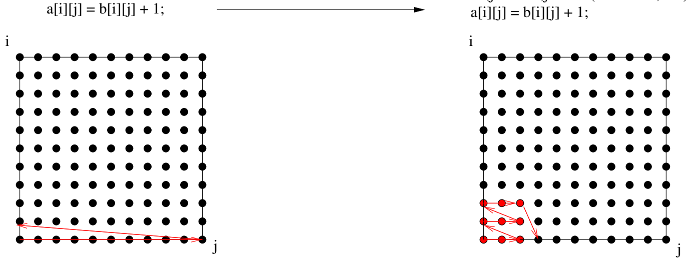
- interchange (for better data locality):

```
for (s1 = 0; s1 <= 15; s1++)
  for (s3 = 0; s3 <= 15; s3++)
    for (i = 64*s1; i <= min (64*s1 + 63, 999); i++)
      for (j = 64*s3; j <= min (64*s3 + 63, 999); j++)
        a[i][j] = b[i][j] + 1;
```

Loop blocking

```
for (i = 0; i < 1000; i++)
  for (j = 0; j < 1000; j++)
    a[i][j] = b[i][j] + 1;
```

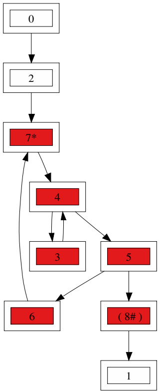
```
for (s1=0;s1<=15;s1++)
  for (s3=0;s3<=15;s3++)
    for (i=64*s1;i<=min(64*s1+63,999);i++)
      for (j=64*s3;j<=min(64*s3+63,999);j++)
        a[i][j] = b[i][j] + 1;
```

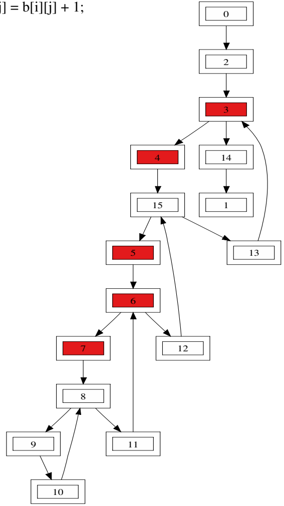


Loop blocking

```
for (i = 0; i < 1000; i++)
  for (j = 0; j < 1000; j++)
    a[i][j] = b[i][j] + 1;
```

```
for (s1=0;s1<=15;s1++)
  for (s3=0;s3<=15;s3++)
    for (i=64*s1;i<=min(64*s1+63,999);i++)
      for (j=64*s3;j<=min(64*s3+63,999);j++)
        a[i][j] = b[i][j] + 1;
```





Loop flattening

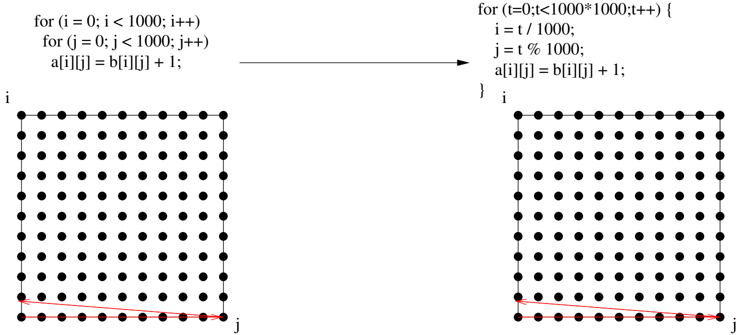
- projection of a loop nest into one dimension (execution trace)
- multiplication of number of iterations for nested loops
- addition of number of iterations for sequential loops
- original loop nest:

```
for (i = 0; i < 1000; i++)
  for (j = 0; j < 1000; j++)
    a[i][j] = b[i][j] + 1;
```

- loop flattening:

```
for (t = 0; t < 1000 * 1000; t++) {
  i = t / 1000;
  j = t % 1000;
  a[i][j] = b[i][j] + 1;
}
```

Loop flattening



same iteration order: loop flattening is always legal

Writing and reading the polyhedral representation

- for GSoC'10 Riyadh Baghdadi added -fgraphite-write and -fgraphite-read to read and write OpenSCoP to disk
- OpenSCoP format: complete polyhedral representation, supported by several other polyhedral tools.
- read and write of OpenSCoP allows development and use of external components (Pluto, PoCC, Pace, etc.)

Auto parallelization with Graphite

- for GSoC'09 Li Feng added -floop-parallelize-all that uses Graphite to tag parallel loops and code generate them using the autopar infrastructure of GCC on top of OpenMP runtime.
- OpenCL code generation: soon to be contributed to Graphite, see the GCCSummit'10 paper "GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops" by Alexey Kravets, Alexander Monakov, and Andrey Belevantsev from Russian Accademy of Science (ISPRAS).
- auto-vectorization on the polyhedral representation: still to be worked on ...