Towards Automatic C Programs Optimization and Parallelization using the PIPS-PoCC Integration

Dounia Khaldi Corinne Ancourt
François Irigoin

CRI, Mathématiques et Systèmes, MINES ParisTech
35 rue Saint-Honoré, 77300. Fontainebleau, France
firstname.lastname@mines-paristech.fr

ABSTRACT

This paper explains how the PIPS source-to-source compilation framework integrates the Polyhedral Compiler Collection (PoCC) as one of PIPS many program transformations. The integration between PIPS and PoCC extracts automatically the static control parts of the source code, which can be optimized independently by PoCC and then reintegrates them transparently in the user source code. PIPS can either be used simply as a wrapper around PoCC to simplify the compilation process, or use PoCC as a step in more complex compilation schemes such as heterogeneous code generation for CUDA or FPGA based-machine. This is possible when the polyhedral model can be used to fine-tune fine-grain parallelism and memory locality. This paper explores the issues in the design of the integration and presents results for some benchmarks of the OpenGPU project.

Keywords

PIPS, PoCC, Optimization, Parallelization, Polyhedral model, Static control code

1. INTRODUCTION AND MOTIVATION

Nowadays, compilers contain a large number of loop optimizations. Several techniques have been proposed to improve cache performance, making effective use of parallel processing capabilities, and reducing overhead associated with control flow. Most execution time of a scientific program is spent on loops. Thus a lot of compiler analysis and compiler optimization techniques have been developed to make the execution of loops faster. To efficiently automate loop transformations, a model is needed on which complex transformations can be easily applied. Transforming the syntax of the loops directly is very cumbersome and makes it difficult to generalize transformations. Therefore, the polyhedral model was developed and transformations are applied to this model. The Polyhedral model is a geometrical representation for programs that utilizes machinery from Linear Algebra and Linear Programming for analysis and high-level transformations [13]. After transformation, efficient code is generated from the polyhedral model, replacing the original loop structure [19]. The scope of the polyhedral model is based on static control parts (SCoP). A regular static control part [10] is a consecutive set of statements with only *'for loops'*, where loop bounds, *'if'* statement conditionals and array accesses are affine functions of the iterators and the global parameters. The ternary operator can be used to provide data-dependent code, where the control will be over-approximated to both clauses being executed for all possible executions. SCoPs are the natural candidates for polyhedral loop transformations. An example is shown in Figure 1.

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++) {
      c[i][j] = 0.0;
          for (k = 0; k < N; k++)
             c[i][j] = c[i][j]+a[i][k]*b[k][j];
       }
```

Figure 1: Example SCoP (matrix multiply)

PoCC [17] is an example of compiler that uses the polyhedral model for loop transformations and parallelization for multicores. Using PoCC, the user has to specify thanks to pragmas where begins the SCoP he is interested by, and where it ends. For the base and development configuration of PoCC, the code fragment inside the pragmas must be a regular static control part. PIPS [8] is a source-to-source compilation framework for analyzing and transforming C and Fortran programs. It is a loop restructuring compiler, implementing polyhedral analyses and transformations. All analyses are interprocedural. PIPS has been used to extract automatically the SCoPs. The goal of this paper is to automate the process of local optimization of SCoPs.

Project OpenGPU aims to take advantage of the computing power of GPUs. It includes an integrated platform to support the parallelization and optimization of existing codes. In this context, the parts of code which could benefit from the architerture characteristics have to be identified, then optimized before being reintegrated back into the user's code. This paper presents two tools PIPS and PoCC used to perform this phase. PIPS processes real and complete applications. It is responsible for identifying the static control parts. PoCC is in charge of the parallelization and optimizations related to the architecture. After transformation, PIPS reintegrates the optimized parts into the initial application.

This paper is organized as follows. Related work in polyhedral compilation is presented in Section 2. Section 3 and 4 describe PoCC and PIPS compilers. Section 5 defines PIPS-PoCC integration. Section 6 presents results and some experiments. Section

7 summarizes the various results.

2. RELATED WORK

Complex tools using the polyhedral model like WRAP-IT for the ORC compiler [5] or the GRAPHITE branch of GCC [4] are devoted to extract static control parts.

In Graphite [20], SCoPs are outlined from the control flow graph. The scalar evolution analysis framework of GCC is used [16]. SCoP outlining scans the basic blocks of the CFG in the dominator order. If a basic block contains a non static control statement then it is considered as difficult and Graphite passes to the next basic block which is dominated by the difficult one. Graphite allows only SCoPs that are surrounded by a single loop. With the option '-graphite-dump-cloog', Graphite dumps each SCoP into a cloog input file.

WRAP-IT extracts automatically SCoPs. It is implemented within the compiler infrastructure Open64/ORC. The output is a list of SCoPs associated with any function in the syntax tree.

The automatic extraction of SCoPs implemented in PIPS, is source to source. The user can observe at any stage of our implementation: Static control code, SCoPs or results of PoCC. SCoP can contain several nested loops. Different optimization options could be selected for the SCoPs. The next figure illustrates the SCoPs extracted by Graphite (on the left handside) and by PIPS (on the right handside) for the program 'gemver.c'. SCoPs extracted by PIPS offer more optimization opportunities such as loop fusion.

```
#pragma scop
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  A[i][j] = A[i][j]+u1[i]*
      v1[j]+u2[i]*v2[j];
#pragma endscop

#pragma scop
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
   x[i]=x[i]+A[j][i]*y[j];
#pragma endscop

#pragma scop
for (i=0; i<N; i++)
 x[i] = x[i] + z[i];
#pragma endscop

#pragma scop
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  w[i] = w[i] +
A[i][j]*x[j];
#pragma endscop
```

```
#pragma scop
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  A[i][j]=A[i][j]+u1[i]
      *v1[j]+u2[i]*v2[j];

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  x[i]=x[i]+A[j][i]*y[j];

for (i=0; i<N; i++)
 x[i]=x[i]+z[i];

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  w[i]=w[i]+A[i][j]*x[j];
#pragma endscop
```

Figure 2: The Extraction of SCoPs by Graphite(left) and by PIPS(right) for 'gemver'

3. PoCC : POLYHEDRAL COMPILER COLLECTION

PoCC [17] is a compiler that uses the polyhedral model for powerful optimization and parallelism. It relies on a set of polyhedral tools in the public domain. The scope of the polyhedral model is based on static control codes or SCoP for short.

PoCC consists in different modules, corresponding to different steps in the optimization process of a source code. It is illustrated in figure 3. PoCC leverages several GNU tools for polyhedral compilation, especially Clan, Candl, LetSee, Pluto and CLooG.

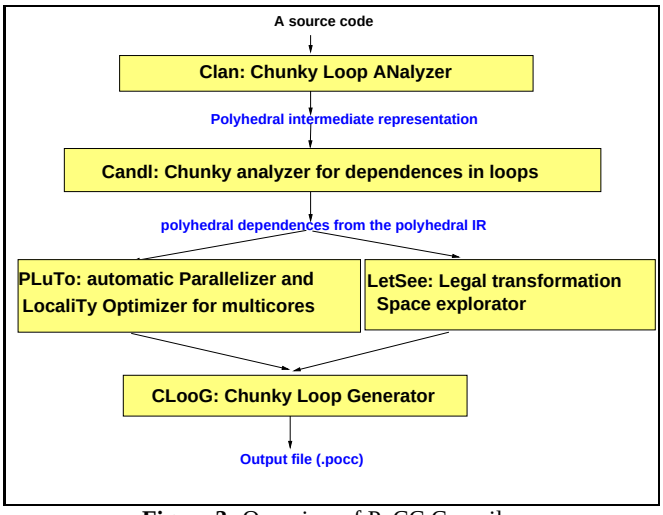

Figure 3: Overview of PoCC Compiler

3.1 Clan and Candl

Clan [10]: The Chunky loop analyzer extracts a polyhedral intermediate representation from the source code of high level programs written in C, C++, C# or Java. Clan considers a given part of the code encapsulated by pragmas #pragma scop and #pragma endscop. For instance, let us consider the following source code in C (Figure 4) of a matrix-matrix multiplication program that reads two matrices, achieves the multiplication and then prints the result. Let us also consider that the user is only interested in the matrix-matrix multiply kernel which is a SCoP. Thus, the user has to add pragmas to this SCoP kernel, so that Clan considers this part of the code.

Candl: The Chunky analyzer for dependences in loops computes polyhedral dependences from the polyhedral IR.

```
/* matmul.c 128*128 matrix multiply */
#include <stdio.h>
#define N 128
int main() {
  int  i,j,k;
  float a[N][N], b[N][N], c[N][N];
  /* We read matrix a then matrix b */
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      scanf("␣%f",&a[i][j]);
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      scanf("␣%f",&b[i][j]);
  /* c = a * b */
#pragma scop
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      c[i][j] = 0.0;
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
#pragma endscop
  /* We print matrix c */
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      printf("%6.2f␣",c[i][j]);
  }
  return 0;
}
```

Figure 4: Pragmas of SCoPs within Matmul program [10]

3.2 PLuTo, LetSee and CLooG

PLuTo [11] is an automatic parallelizer and locality optimizer for multicores. It is based on the polyhedral model. PLuTo transforms C programs from source to source for coarse-grained parallelism and locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but is not limited to it. It performs prevectorization, scalar privatization, array contraction and many loop transformations such as: loop fusion, loop unroll. OpenMP parallel code for multicores can be automatically generated from sequential C program sections.

LetSee [18], the Legal transformation Space explorator, is a platform dedicated to computing and exploring the legal affine scheduling of a program. It can be used to optimize a program, the output being an optimized C code corresponding to the polyhedral representation given as an input.

CLooG [9], the Chunky Loop Generator is a code generator in the polyhedral model. It is also used to avoid overhead associated with control flow and to produce effective code.

4. PIPS : AUTOMATIC PARALLELIZER AND CODE TRANSFORMATION FRAMEWORK

PIPS [8] is a source-to-source compilation framework for analyzing and transforming C and Fortran programs. PIPS implements polyhedral analyses and transformations [1]. All analyses are interprocedural. Figure 5 resumes its features like:

Use-def chains, data dependence graph, transformers, preconditions, symbolic complexity, memory effects (of instructions on data), convex array regions, call graph, interprocedural control flow graph...

Many program transformations can be applied, such as:

- parallelization, several algorithms are implemented:
  - Allen & Kennedy's parallelization algorithm that may perform some loop distribution.
  - Vectorization: selects innermost loops for vector units.
  - Coarse grain parallelization: is based on the convex array region of the loops, no loop distribution is performed.
- scalar and array privatization: consists in discovering variables whose values are local to a particular scope, usually a loop iteration.
- loop transformations: loop unrolling, interchange, normalization, distribution, strip mining, tiling, index set splitting.
- dead-code elimination, partial evaluation, atomization, control restructuration, loop invariant code motion, forward substitution...
- inlining, outlining, cloning of functions

PIPS generates parallel code for MMX, SSE, CUDA ... architectures [2]. It generates OpenMP code and tranforms code from OpenMP to MPI.

Pipsmake library [1] manages objects that are resources stored in memory or/and on disk. The phases are described by generic rules, which use and produce resources. Examples of these rules are illustrated in the following section.

[1] `http://cri.ensmp.fr/PIPS/pipsmake.html`

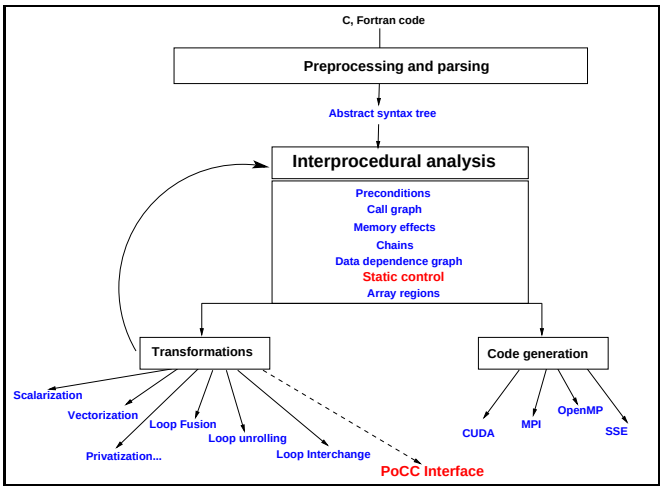

Figure 5: Overview of PIPS Compiler with its PoCC Interface (dashed arrow)

5. PIPS-PoCC INTERFACE

We present the steps that have been implemented in PIPS to optimize/parallelize every SCoP in a program. These steps are listed in the command line interpreter of PIPS: tpips. Figure 6 summarizes these steps: Static control detection, SCoP outlining, use of PoCC and SCoP inlining. Figure 7 illustrates matmul.tpips. It is the script to use to execute our integration.

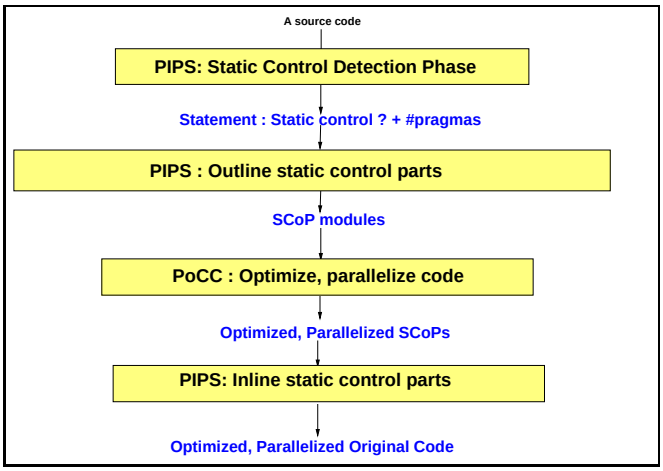

Figure 6: PIPS-PoCC Integration

5.1 Static Control Detection

The static_controlize phase gets the structural parameters. It detects for each statement enclosing loops, enclosing tests and the static_control property. Those informations are mapped on statements. The definition of a static control program is given in [12]. It is a program in which all loops are *'for'* loops whose limits depend only on structure parameters, numerical constants and outer loops iteration counters.

For this pass, the code and the list of variables are needed to generate the 'static_control' resource of a given module (or function)

```
Static_Controlize   >  MODULE.static_control
< PROGRAM.entities
< MODULE.code
```

For instance, let us consider the matrix-matrix multiply program, the result of the 'Static_Controlize' phase is illustrated on Figure 7.

5.2 Automatic SCoP Extraction and Outlining

Static control parts (SCoPs) are outlined after application of the 'static controlize' phase. For this phase, the module and static control informations and program variables are used to outline SCoPs of a given module.

```
#Create a workspace for the program matmul.c
create matmul matmul.c
#Compute SCoPs and encapsulate them with pragmas
apply POCC_PRETTYPRINTER[main]
#Outline SCoPs
apply SCOP_OUTLINER[main]
setenv list_pocc matmul.database/*SCoP_*/*SCoP_*.c
#Set PoCC optimization options
setenv PoCC_FLAGS "--pluto-unroll"
#Application of PoCC on each SCoP
shell for i in list_pocc; do \
        pocc PoCC_FLAGS $i; done
setenv list_pocc 'echo $list_pocc_c|sed
                  's/matmul.database\///g''
#Reinline results of PoCC
apply INLINING[$list_pocc]
#Print the result on stdout
display PRINTED_FILE[main]
#generate the entire file(matmul.c)
apply UNSPLIT[main]
#compile it
shell cd matmul.database/Src ; gcc -c matmul.c
close
quit
```

Figure 7: Executable (matmul.tpips) for matmul.c

```
   /* c = a * b */

   for(i = 0; i <= 127; i += 1)
//              < is static > TRUE
//              < parameter > k, j, i,
//              <   loops   > 0 <= i <= 127
//              <   tests   >
      for(j = 0; j <= 127; j += 1) {
//              < is static > TRUE
//              < parameter > k, j, i,
//              <   loops   > 0 <= i <= 127
//                            0 <= j <= 127
//              <   tests   >
         c[i][j] = 0.0;
//              < is static > TRUE
//              < parameter > k, j, i,
//              <   loops   > 0 <= i <= 127
//                            0 <= j <= 127
//              <   tests   >
         for(k = 0; k <= 127; k += 1)
//              < is static > TRUE
//              < parameter > k, j, i,
//              <   loops   > 0 <= i <= 127
//                            0 <= j <= 127
//                            0 <= k <= 127
//              <   tests   >
            c[i][j] = c[i][j]+a[i][k]*b[k][j];
      }
//              < is static > TRUE
//              < parameter > k, j, i,
//              <   loops   >
//              <   tests   >
```

Figure 8: Static control informations for Matmul program

```
Scop_Outliner        >       MODULE.code
     > MODULE.callees
     < PROGRAM.entities
     < MODULE.code
     < MODULE.static_control
```

After outlining, additional functions are generated. The set of module callees (> MODULE.callees) has been changed. Figure 9 shows the result of the 'SCoP_Outliner' phase for the matrix-matrix multiply program. Figure 10 shows the SCoP code outlined by the 'SCoP_Outliner' phase which represents the matrix multiplication kernel. The result of the outlining is a set of modules prefixed by 'SCoP_'.

```
int main()
{
   int i, j, k;
   float a[128][128], b[128][128], c[128][128];

   /* We read matrix a */
   for(i = 0; i <= 127; i += 1)
      for(j = 0; j <= 127; j += 1)
         scanf("␣%f", &a[i][j]);
   /* We read matrix b */
   for(i = 0; i <= 127; i += 1)
      for(j = 0; j <= 127; j += 1)
         scanf("␣%f", &b[i][j]);
   /* c = a * b */

   SCoP_0(a, b, c);

   /* We print matrix c */
   for(i = 0; i <= 127; i += 1) {
      for(j = 0; j <= 127; j += 1)
         printf("%6.2f␣", c[i][j]);
      printf("\n");
   }
   return 0;
}
```

Figure 9: Matmul after outlining of one SCoP

```
void SCoP_0(float a[128][128], float b[128][128],
                               float c[128][128])
{
   //PIPS generated variable
   int i, j, k;
   /* c = a * b */
#pragma  scop
   for(i = 0; i <= 127; i += 1)
      for(j = 0; j <= 127; j += 1) {
         c[i][j] = 0.0;
         for(k = 0; k <= 127; k += 1)
            c[i][j] = c[i][j]+a[i][k]*b[k][j];
      }
#pragma  endscop

   return;
}
```

Figure 10: The outlined matrix-matrix multiply kernel

5.3 Application of PoCC on SCoPs

The next step is the PoCC optimization. Many options such as unroll, parallelize, prevectorize can be chosen. Modules with extension '.pocc.c' are generated. Figure 11 shows the result of execution of PoCC on 'SCoP_0' with the option '- -pluto-tile - - pluto-parallel- -pluto-unroll'. The optimizer PLuTo is activated with the options polyhedral Loop tiling and loop unrolling. '- -pluto-tile' partitions the loop's iteration space into 4 blocks of 32, to ensure

Table 1: (Static Control loop nests) / (all loop nests in the application) [in %]

Benchmark	SCoPs	Ratio of SCoPs / loop nests	Comments
STAP	21 SCoPs	65%	Some upper bounds of loops are not linear
ABF	15 SCoPs	100%	All loop nests are SCoPs
FMradio	9 SCoPs	90%	One loop nest bound contains pointer access

data locality in cache. '- - pluto-parallel' generates the OpenMP code. '- -pluto-unroll' proceeds with unroll and jam (ufactor=4) for loop 'j', and with unroll (ufactor=4) for Loop 'k'. PoCC has made decisions about unrolling such as: not unroll Loop 'i' to not explose the code size. Tile sizes can be specified in a file 'tile.sizes', otherwise default sizes will be set.

5.4 SCoP Inlining

Inlining is a well-known technique. It is implemented in PIPS. It replaces a function call by the function body. In our context, all function calls to SCoPs, that were transformed/parallelized by PoCC, are inlined. The goal is to offer the user the structure of its original program.

6. EXPERIMENTS

In this section, we present the results of PIPS_PoCC integration on signal processing applications STAP, ABF and FMradio.

6.1 Overview of the Applications

Space-time adaptive processing (STAP) [15] application has been developed in Thales. It is a powerful method to remove the ambiguity consisting in computing from the signal received both from different antenna (or sub-arrays) and at different times a set of filters that will permit to make the distinction.
Adaptive Beam Forming (ABF) is a beamforming system which performs adaptive spatial signal processing with an array of radar antennas (or phased array) in order to transmit or receive signals in different directions without having to mechanically steer the array.
FMradio is a kernel extracted from the GNU Radio project [6].
STAP and ABF were slightly modified in order to be optimized by PoCC (the structure of complex floats is changed into an array of floats).

6.2 Using the PIPS-PoCC Integration

PIPS parses and analyses the applications. It extracts static control parts and outlines them. SCoPs are generated in new modules prefixed by 'SCoP_'. The resulting codes can be optimized, parallelized by PoCC compiler. Table 1 shows the results of our integration on the signal processing applications STAP, ABF and FMradio. The ratio represents the number of static control loop nests / all loop nests in the application.
Performance results of PoCC are presented at [3].

7. CONCLUSION

We present an integration between PIPS and PoCC that automatizes parallelization and local optimizations in the polyhedral model. First, PIPS detects static control code, and outlines SCoPs. Next, PoCC optimizes/parallelizes SCoPs. Finally, PIPS inlines SCoPs and gives back original structure to user's program. In this paper we have selected PoCC compiler that locally optimizes functions, but our implementation can be a front end for other compilers.
PIPS-PoCC integration is a transformation in PIPS that can serve as an optimization phase for many compilers such as the two automatic compilers for the C language, terapyps [14] and p4a [7], for an **FPGA** based embedded processor and nvidia **GPU**, respectively. Figure 12 illustrates this feature.

8. ACKNOWLEDGMENT

This work is funded by the OpenGPU Project. `http://opengpu.net/`.
The authors would like to thank Cédric Bastoul and Pierre Jouvelot for their comments on this paper and to Antoniu Pop for his help to install Graphite. Many thanks also to all PIPS contributors and especially to Arnaud Leservot who implemented the static_controlize phase in PIPS.

9. REFERENCES

[1] Mines paristech. pips. `http://pips4u.org`, 1989-2011. Open source, under GPLv3.
[2] Hpc project. par4all. `http://www.par4all.org`, 2008.
[3] Performance results of pocc. `http://www-rocq.inria.fr/~pouchet/software/pocc/doc/htmldoc/htmldoc/index.html`.
[4] Graphite: Gimple represented as polyhedra. `http://gcc.gnu.org/wiki/Graphite`.
[5] Wrap-it: Whirl represented as polyhedra. `http://www.lri.fr/~girbal/site_wrapit`.
[6] The gnu radio project. `http://www.gnu.org/software/gnuradio/`.
[7] M. Amini, F. Irigoin, and R. Keryell. Optimisation statique des communications hôte-accélérateur dans un paralléliseur automatique. Technical Report A/451, CRI, Mathématiques et Systèmes, MINES-ParisTech, Fontainebleau, France, 2010.
[8] C. Ancourt, B. Breusillet, F. Coelho, F. Irigoin, P. Jouvelot, and R. Keryell. Pips a workbench for interprocedural program analyses and parallelization. In *In Meeting on data parallel languages and compilers for portable parallel computing*, 1994.
[9] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.
[10] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC 16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, Texas, october 2003.
[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical and automatic polyhedral program optimization system. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, Tucson, June 2008.
[12] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, pages 20(1):23–53, February 1991.
[13] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of

```
void SCoP_0(float a[128][128], float b[128][128],
        float c[128][128])
{
 lb1=0; ub1=3;
#pragma omp parallel for shared(t1,lb1,ub1)
        private(t2,t3,t4,t5,t6,k)
for (t2=lb1; t2<=ub1; t2++) {
  for (t3=0;t3<=3;t3++) {
    for (t5=32*t2;t5<=32*t2+31;t5++) {
      lbv=32*t3; ubv=32*t3+31;
#pragma ivdep
#pragma vector always
      for (k=lbv; k<=ubv; k++) {
        c[t5][k]=0.0;
      }
    }}}
 lb1=0; ub1=3;
#pragma omp parallel for shared(t1,lb1,ub1)
        private(t2,t3,t4,t5,t6,k)
 for (t2=lb1; t2<=ub1; t2++) {
   for (t3=0;t3<=3;t3++) {
     for (t4=0;t4<=3;t4++) {
       for (i=32*t2; i<=32*t2+28; i=i+4) {
         for (j=32*t4; j<=32*t4+28; j=j+4) {
           lbv=32*t3; ubv=32*t3+31;
#pragma ivdep
#pragma vector always
 for (k=lbv; k<=ubv; k++) {
   c[i][k]=c[i][k]+a[i][j]*b[j][k];
   c[i][k]=c[i][k]+a[i][(j+1)]*b[(j+1)][k];
   c[i][k]=c[i][k]+a[i][(j+2)]*b[(j+2)][k];
   c[i][k]=c[i][k]+a[i][(j+3)]*b[(j+3)][k];
   c[(i+1)][k]=c[(i+1)][k]+a[(i+1)][j]*b[j][k];
   c[(i+1)][k]=c[(i+1)][k]+a[(i+1)][(j+1)]*b[(j+1)][k];
   c[(i+1)][k]=c[(i+1)][k]+a[(i+1)][(j+2)]*b[(j+2)][k];
   c[(i+1)][k]=c[(i+1)][k]+a[(i+1)][(j+3)]*b[(j+3)][k];
   c[(i+2)][k]=c[(i+2)][k]+a[(i+2)][j]*b[j][k];
   c[(i+2)][k]=c[(i+2)][k]+a[(i+2)][(j+1)]*b[(j+1)][k];
   c[(i+2)][k]=c[(i+2)][k]+a[(i+2)][(j+2)]*b[(j+2)][k];
   c[(i+2)][k]=c[(i+2)][k]+a[(i+2)][(j+3)]*b[(j+3)][k];
   c[(i+3)][k]=c[(i+3)][k]+a[(i+3)][j]*b[j][k];
   c[(i+3)][k]=c[(i+3)][k]+a[(i+3)][(j+1)]*b[(j+1)][k];
   c[(i+3)][k]=c[(i+3)][k]+a[(i+3)][(j+2)]*b[(j+2)][k];
   c[(i+3)][k]=c[(i+3)][k]+a[(i+3)][(j+3)]*b[(j+3)][k];
 }
           }
           for (t6=j; t6<=32*t4+31; t6=t6+1) {
             lbv=32*t3; ubv=32*t3+31;
#pragma ivdep
#pragma vector always
 for (k=lbv; k<=ubv; k++) {
   c[i][k]=c[i][k]+a[i][t6]*b[t6][k];
   c[(i+1)][k]=c[(i+1)][k]+a[(i+1)][t6]*b[t6][k];
   c[(i+2)][k]=c[(i+2)][k]+a[(i+2)][t6]*b[t6][k];
   c[(i+3)][k]=c[(i+3)][k]+a[(i+3)][t6]*b[t6][k];
 }
 }}
  for (t5=i; t5<=32*t2+31; t5=t5+1) {
    for (j=32*t4; j<=32*t4+28; j=j+4) {
      lbv=32*t3; ubv=32*t3+31;
#pragma ivdep
#pragma vector always
 for (k=lbv; k<=ubv; k++) {
   c[t5][k]=c[t5][k]+a[t5][j]*b[j][k];
   c[t5][k]=c[t5][k]+a[t5][(j+1)]*b[(j+1)][k];
   c[t5][k]=c[t5][k]+a[t5][(j+2)]*b[(j+2)][k];
   c[t5][k]=c[t5][k]+a[t5][(j+3)]*b[(j+3)][k];
 }}
    for (t6=j; t6<=32*t4+31; t6=t6+1) {
      lbv=32*t3; ubv=32*t3+31;
#pragma ivdep
#pragma vector always
      for (k=lbv; k<=ubv; k++) {
        c[t5][k]=c[t5][k]+a[t5][t6]*b[t6][k];
      }
    }
  }
    }
   }
 }
#pragma endscop
  return;
}
```

Figure 11: PoCC - -pluto-tile - -pluto-parallel - -pluto-unroll SCoP_0.c

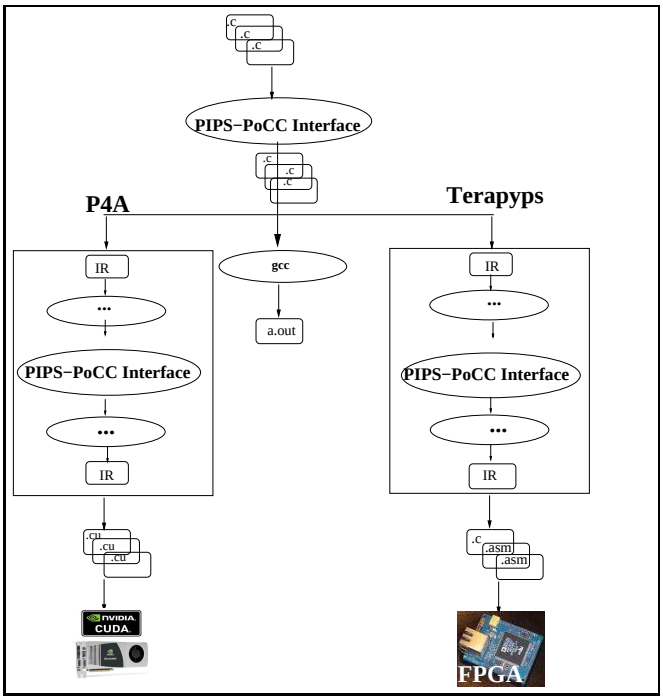

Figure 12: Application of PIPS-PoCC Integration

loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[14] S. Guelton, R. Keryell, and F. Irigoin. Compilation for heterogeneous computing: Automating analyses, transformations and decisions. Technical Report A/450, CRI, Mathématiques et Systèmes, MINES-ParisTech, Fontainebleau, France, 2010.

[15] F. Le Chevalier, M. Montecot, Y. Doisy, F. Letestu, and P. Chevalier. Stap developments in thales. In *Radar Conference, 2009. EuRAD 2009. European Issue*, pages 53 – 56, sept 2009.

[16] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, pages 218–232. Springer-Verlag, Nov 2005.

[17] L.-N. Pouchet and C. Bastoul. *PoCC, The Polyhedral Compiler Collection package Edition 0.3, for PoCC 1.0-rc2*, April 13rd 2010.

[18] L.-N. Pouchet, C. Bastoul, and A. Cohen. Letsee: the legal transformation space explorator. In *Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, L'Aquila, Italia, july 2007.

[19] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program*, pages 28(5):469– 498, 2000.

[20] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa Italy, 01 2010.