PIPS Is not (just) Polyhedral Software

Adding GPU Code Generation in PIPS

Mehdi AMINI[1,3] Corinne ANCOURT[1] Fabien COELHO[1] Béatrice CREUSILLET[3] Serge GUELTON[2] François IRIGOIN[1] Pierre JOUVELOT[1] Ronan KERYELL[3] Pierre VILLALON[3]

[1]MINES ParisTech *firstname.lastname*@mines-paristech.fr [2]Télécom Bretagne *firstname.lastname*@telecom-bretagne.eu
[3]HPC Project *firstname.lastname*@hpc-project.com

Abstract

Parallel and heterogeneous computing are growing in audience thanks to the increased performance brought by ubiquitous many-cores and GPUs. However, available programming models, like OPENCL or CUDA, are far from being straightforward to use. As a consequence, several automated or semi-automated approaches have been proposed to automatically generate hardware-level codes from high-level sequential sources.

Polyhedral models are becoming more popular because of their combination of expressiveness, compactness, and accurate abstraction of the data-parallel behaviour of programs. These models provide automatic or semi-automatic parallelization and code transformation capabilities that target such modern parallel architectures.

PIPS is a quarter-century old source-to-source transformation framework that initially targeted parallel machines but then evolved to include other targets. PIPS uses abstract interpretation on an integer polyhedral lattice to represent program code, allowing linear relation analysis on integer variables in an interprocedural way. The same representation is used for the dependence test and the convex array region analysis. The polyhedral model is also more classically used to schedule code from linear constraints.

In this paper, we illustrate the features of this compiler infrastructure on an hypothetical input code, demonstrating the combination of polyhedral and non polyhedral transformations. PIPS interprocedural polyhedral analyses are used to generate data transfers and are combined with non-polyhedral transformations to achieve efficient CUDA code generation.

Keywords Heterogeneous computing, convex array regions, source-to-source compilation, polyhedral model, GPU, CUDA, OPENCL.

1. Introduction

Parallelism has been a research subject since the 50's and a niche market with vector machines, since the 70's, and parallel machines, since the 80's. Today, parallelism is the only viable option to achieve higher performance with lower electrical consumption. The trend in current architectures is toward higher parallelism and heterogeneity in all application domains, from embedded systems up to high-end supercomputers.

To ease vector machine programming, automatic vectorizers were developed in the 80's. The PIPS project began just 23 years ago: it was a source-to-source Fortran infrastructure designed to be independent of the target language. It used an abstract interpretation based on a polyhedra lattice with sparse linear algebra support to achieve interprocedural parallelization of ECG signal processing and of scientific code [24, 34].

With the development of cheaper and more powerful parallel computers in the 80's as an alternative to vector computers, automatic parallelization became an important and highly-investigated research topic, and was incorporated into PIPS. Automatic parallelization in PIPS also relied on polyhedral and linear algebra foundations, a specific mathematical approach developed in France.

However, due to the overly high expectations on automatic vectorization and parallelization to solve the difficulties of high-performance programming, the domain was deserted after decades of research. Some teams have been in the polyhedral core development from the beginning with an interest in parallelization [34] and code analysis [12]. This has evolved into different research streams including polyhedral models, which are exact representations of restricted programs to give optimal solutions [17], approximate representations, which compile real software as a whole [24], and theorem-proving frameworks, that reason about facts and properties of programs [19].

After 20 years of mostly divergent developments, the three approaches are beginning to converge, not only because of the complementary characteristics and cross-dependence that they exhibit in their maturity but also because the growth of the parallel machine market has prompted a return to the roots of these research thrusts: solving real programming issues on real machines.

Compiling and optimizing programs both globally and locally, across and within kernels and for even more complex heterogeneous parallel architectures, is now a challenge requiring the development of sophisticated tools, which is quite difficult to start from scratch. Therefore, the community has concentrated on building specific tools for the difficult sub-problems, with gateways between tools to create synergy. Many of such compiling infrastructures use or used the polyhedral model internally at different degrees: GCC with GRAPHITE [33], POCC that integrates various tools [32] such as Pluto [5] or CLOOG [6], subprojects in ROSE [15] and in LLVM [26] that will include POCC support, SUIF [20], as well as some of the compilers offered by vendors.

Since many efforts now focus on heterogeneous code generation, in this paper we present PIPS with its prior accomplishments but also with this new perspective. Tools like Pluto, CLOOG or GRAPHITE explicitly use the polyhedral model to represent and to optimize further loop nests, whereas PIPS relies on a hierarchical control flow graph and interprocedural analysis that gathers, propagates and accumulates results along this graph.

This alternative approach is presented in the paper, and organized as follows: Section 2 first describes some of PIPS use cases; Section 3 defines key PIPS analyses used in the context of heterogeneous code generation. The transformations applied to automatically generate CUDA code on an example are finally presented in Section 4.

2. Key Use Cases

This section contains examples of PIPS usage to show the power of source-to-source compilation coupled with a strong polyhedral and linear algebra-based abstract interpretation framework to solve a wide range of problems.

2.1 Vectorization and Parallelization

The meaning of PIPS at the very beginning was *Parallélisation interprocédurale de programme scientifiques* or Interprocedural Parallelization of Scientific Programs and was targeting the vector supercomputers from the 80's such as Cray by translating Fortran 77 to Cray Fortran with directives. This project introduced the interprocedural parallelization based on linear algebra method [13, 14, 24, 34, 35]. Later other parallel languages and programming models have been added to express parallelism for various parallel computers: CMFortran for the Connection Machine, Fortran 90 parallel array syntax, HPF parallel loops and OPENMP parallel loops.

More recently, this historical subject has again become relevant with the development of SIMD vector instructions in almost every processor from embedded systems (smart phones) up to high-end computers to improve their energy efficiency. The generation of vector instruction intrinsic functions such as SSE or AVX for x86 and Neon for ARM processors has been done in the SAC [18] PIPS subproject. Code generation for the CUDA and OPENCL vector types is on-going to improve further the efficiency of the generated code for these targets.

2.2 Code and Memory Distribution

For a Transputer-based computer, automatic code parallelization and distribution of sequential code have been developed. Different programs were generated for compute nodes and for memory nodes in order to emulate a global shared memory. This project introduced code generation by scanning polyhedra [2] and code distribution with a linear algebra method [1].

More recently, PIPS has been used to generate SPMD MPI programs from OPENMP annotated code [29] by using PIPS convex array regions.

2.3 HPF Compilation

The previous method was extended and transposed to the High Performance Fortran world where the alignment and distribution methods were affine. A polyhedral method was used to distribute code and data on nodes with a distributed memory and to generate the proper communications [3]. The communications, I/O and data remapping were also optimized [8, 9].

2.4 Compilation for Heterogeneous Targets

The development of heterogeneous computing with accelerators and complex programming models motivates providing higher level tools, for example the direct compilation of sequential programs. The previous techniques have been adapted to generate from sequential code annotated with pragmas host and accelerator code for the CoMap FPGA pipelined accelerator [21]. This approach has been generalized and improved to suite the Ter@pix vector accelerator. Support for the CEA SCMP accelerator is on-going.

The generation of configurations for the SPOC configurable image processing pipeline has been done to illustrate the use on non classical accelerators from sequential code [10].

The PAR4ALL project [22] uses these techniques to generate CUDA code for NVIDIA GPU and this is currently extended for OPENCL to target other GPU and embedded systems such as ST Microelectronics P2012.

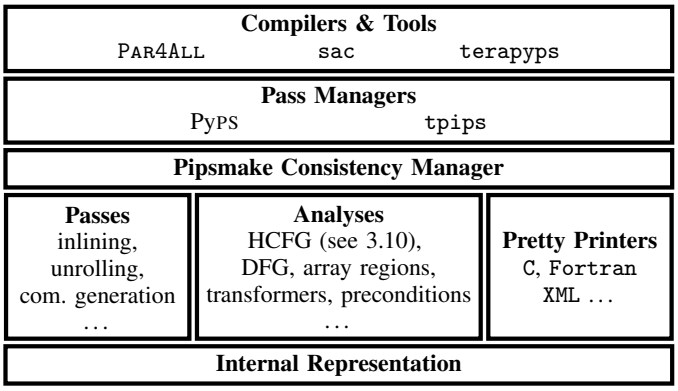

Figure 1: PIPS infrastructure.

2.5 Program Verification

Since automatic parallelization and abstract interpretation in PIPS use verifiers of mathematical polyhedral proofs, they can be used to extract semantics properties in order to prove facts about programs. For example, this is used to perform array bound checking and remove provably redundant array bound checks [30]. It is currently extended to extract more precise linear integer pre- and postconditions on programs.

2.6 Program Synthesis

Code generation from specifications has been combined with the SPEAR-DE [27] and GASPARD [16] tools to generate actual signal processing code with allocation and communication based on array regions. Now development is underway in PIPS to compose Simulink, Scade or Xcos/Scicos components by analyzing the C code of components.

2.7 High-level Hardware Synthesis

From sequential code, we have implemented for the PHRASE FPGA reconfigurable accelerator a generator for the Madeo hardware description language based on SmallTalk [7].

2.8 Decompilation and Reverse Engineering

PIPS has also been used at HPC Project after a home brewed binary disassembler to regenerate high level C code with loops to perform direct parallelization from binary executables by using linear information found on the code.

More classically, PIPS analyses results can help to understand and maintain code. Transformations can be used too, for instance when key variables have constant values because a simplified version of a code is requested. As an example, a 3D code can be automatically transformed into a 2D code when only the 2-D functionality is useful so as to reduce maintenance cost. Furthermore, some refactoring can be performed by simplifying control structures and eliminating useless local and global variables and by refining array declarations. Array bound checks may provide additional information in maintenance.

3. Key PIPS Internals

Figure 1 presents the global organization of the PIPS infrastructure. A typical compilation scheme involves a front-end to split input files in modules, one per function, and generate PIPS hierarchical internal representation for each of them. Then a transformation is called on a specific module by the pass manager. The consistency manager takes care of calling the appropriate analyses. Such analy-

Listing 1: Sample cross correlation of two length-N signals.

```
float corr_body(int k,int N,float x[N],float y[N]){
  float out = 0.;int n=N-k;
  while(n>0) {
    n = n - 1;
    out+=x[n]*y[n] / N;
  }
  return out;
}
int corr(int N,float x[N],float y[N],
         int M,float R[M]){
  if( M < N ) {
    for(int k=0; k<M; k++)
      R[k]=corr_body(k, N, &x[k], y);
    return 1;
  }
  else
    return 0;
}
```

ses may be interprocedural. In that case data collection is a two-step process: first proper information is computed intra-procedurally, then the data is aggregated interprocedurally. If need be, this process can be performed iteratively in order to gather more accurate results. Finally the transformation is fed with the analysis result and applies its algorithm.

In the following we survey a few key features of the around 300 phases and analyses available in PIPS. The signal processing code sample in Listing 1, adapted from [31], is used throughout the article to illustrate the corresponding transformations. All the presented codes are the result of running PIPS on it, *modulo* cosmetic changes.

3.1 Simple Memory Effects

The main first analyses scheduled are the *simple effects* analyses, which describe the memory operations performed by a given statement. During these analyses, arrays are considered atomically. *Proper effects* are memory references local to individual statements. *Cumulated effects* take into account all effects of compound statements, including those of their sub-statements. *Summary effects* summarize the cumulated effects for a function and mask effects on local entities. Effects are used to give the *def/use* information on function parameters.

Many analyzes we'll introduce later rely on these base analyses. They are used by the *outlining* phase to distinguish between parameters passed by values (only read) or by reference (written), as shown in Figure 2. Memory effects on Listing 2 shows which arrays are involved in the computation, and tells us that both `M`, `N` and `k` are only read. It leads to the outlining presented in Listing 3.

3.2 Transformers

The *transformers* are the basis for *linear relation analysis* [12] in PIPS and represent in a linear way the relation between the store after an instruction and the store before [23], mainly for integer variables. Some information is also kept for float, complex, boolean and string variables. Transformers can also be interpreted as polyhedra with both the values before and after representing their relations.

Listing 4 illustrates this analysis on the example from Listing 1, with transformers displayed as comments before each statement, with special notation `n'` representing the value of Variable `n` *after* the statement. For instance, the iteration over Variable `n` is represented by the transformer of the while loop body, namely `n'==n-1`; this information is used to recover "for" loops from while loops, resulting in Listing 5.

Listing 2 : Cumulated effects analysis.

```
//      <may be read >: x[*] y[*]
//      <may be written>: R[*]
//      <  is read >: M N
int corr(int N,float x[N],float y[N],
         int M,float R[M]){
//      <may be read >: x[*] y[*]
//      <may be written>: R[*]
//      <  is read >: M N
 if (M<N) {{
//      <may be read >: N k x[*] y[*]
//      <may be written>: R[*]
//      <  is read >: M
//      <  is written>: k
   for(int k = 0; k <= M-1; k += 1)
//      <may be read >: x[*] y[*]
//      <may be written>: R[*]
//      <  is read >: M N k
     R[k] = corr_body(k,N,&x[k],y);
  }
  return 1;
 }
 else
  return 0;
}
```

Listing 3 : Outlining illustration.

```
int corr(int N,float x[N],float y[N],
         int M,float R[M]){
 if (M<N) {{
   for(int k = 0; k <= M-1; k += 1)
    kernel(M,N,k,R,x,y);
  }
  return 1;
 }
 else
  return 0;
}
void kernel(int M,int N,int k,float R[M],
            float x[N],float y[N]){
 R[k] = corr_body(k,N,&x[k],y);
}
```

Figure 2: Cumulated effects and their usage in outlining.

3.3 Preconditions

Preconditions [23] are affine predicates over scalar variables. It is a predicate always true before the execution of the statement it is attached to. It is an interprocedural analysis; preconditions are propagated downwards on the abstract syntax tree leaves by combination with the transformers.

At each call site, the callee's transformer is used to propagate the preconditions in the caller and the call site precondition is unioned with other call site preconditions to generate the callee's *summary precondition*. Thanks to this summarization, each function is analyzed only once. These predicates are used as input, each time symbolic constants or expressions have to be evaluated: lower and upper bounds, strides, offset, subscript... E.g. Listing 6 displays interprocedural propagation of preconditions in the case of our running example.

Using PIPS's pass manager, it is possible to improve the accuracy of this analysis by taking already calculated preconditions into account. This leads to an iterative analysis described in [4].

3.4 Convex Array Regions

Convex Array Regions [13, 14] abstract with affine equalities and inequalities the set of array elements accessed during the execution of a given section of code. The characteristics on the referenced array elements are picked from Regions. Then an approximation of the amount of memory required to allocate task arrays can

Listing 4 : Result of transformers analysis.

```
//  T() {}
float corr_body(int k,int N,float x[N],float y[N]){
//  T() {}
 float out = 0.;
//  T(n) {k+n'==N}
 int n = N-k;
//  T(n) {k+n==N,1<=n',n'<=n,1<=n}
 while (n>0) {
//  T(n) {n'==n-1,k+1<=N,0<=n'}
   n = n-1;
//  T() {k+1<=N,0<=n}
   out += x[n]*y[n]/N;
 }
//  T() {k+n<=N,n<=0}
 return out;
}
```

Listing 5 : After "for" loops recovery.

```
float corr_body(int k,int N,float x[N],float y[N]){
 float out = 0.;
 int n = N-k;

 for(int n0 = n; n0 >= 1; n0 += -1) {
  n = n0 -1;
  out += x[n]*y[n]/N;
 }
 return out;
}
```

Figure 3: Illustration and usage of the transformer analysis.

Listing 6: Result of interprocedural preconditions analysis.

```
//  P() {k+2<=N,0<=k}
float corr_body(int k,int N,float x[N],float y[N]){
//  P() {k+2<=N,0<=k}
 float out = 0.;
//  P() {k+2<=N,0<=k}
 int n = N-k;
//  P(n) {k+n==N,k+2<=N,0<=k}
 while (n>0) {
//  P(n) {k+2<=N,k+n<=N,0<=k,1<=n}
   n = n-1;
//  P(n) {k+2<=N,k+n+1<=N,0<=k,0<=n}
   out += x[n]*y[n]/N;
 }
//  P(n) {n==0,k+2<=N,0<=k}
 return out;
}
```

be deduced. The similarity of concept between array regions and data transfer between a host and a remote accelerator is used to automatically generate the latter in PAR4ALL. Listing 7 displays the result of this analysis on the code from Listing 1.

3.5 Complexities

Complexities [36] are symbolic estimations of the execution time of statements. They are computed interprocedurally and based on polynomial models of execution time. The complexity is less precise in case of tests, but it gives a magnitude order of task operations (symbolic or/and numeric).

3.6 Dependence Tests

Several dependence tests [35] are implemented in PIPS. They compute dependence cones or levels. Dependences on loops are used for parallelization but also to help the various mapping tools.

Listing 7: Result of interprocedural convex array region analysis.

```
//  <R[PHI1]-W-MAY-{0<=PHI1,PHI1+1<=M,M+1<=N}>
//  <x[PHI1]-R-MAY-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
//  <y[PHI1]-R-MAY-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
int corr(int N,float x[N],float y[N],
         int M,float R[M]){
//  <R[PHI1]-W-MAY-{0<=PHI1,PHI1+1<=M,M+1<=N}>
//  <x[PHI1]-R-MAY-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
//  <y[PHI1]-R-MAY-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
 if (M<N) {{
//  <R[PHI1]-W-EXACT-{0<=PHI1,PHI1+1<=M,M+1<=N}>
//  <x[PHI1]-R-EXACT-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
//  <y[PHI1]-R-EXACT-{0<=PHI1,PHI1+1<=N,1<=M,M+1<=N}>
   for(int k = 0; k <= M-1; k += 1)
//  <R[PHI1]-W-EXACT-{PHI1==k,0<=k,k+1<=M,M+1<=N}>
//  <x[PHI1]-R-EXACT-{k<=PHI1,PHI1+1<=N,0<=k,k+1<=M,M
    +1<=N}>
//  <y[PHI1]-R-EXACT-{0<=PHI1,PHI1+k+1<=N,0<=k,k+1<=M
    ,M+1<=N}>
    kernel(M,N,k,R,x,y);
  }
  return 1;
 }
 else
  return 0;
}
```

3.7 Linear Algebra for Analyses and Transformations

PIPS analyses are based on the C^3 linear algebra library, mainly developed at MINES ParisTech from the 80's. It provides classical types (integer vectors, matrix, polynomial...) with various algorithms (mathematical operations, HERMITE's normal form, SMITH's normal form, sorting, simplex...) allowing the implementation of all the PIPS polyhedral and linear analyses and transformations, such as unimodular transformations. Our interprocedural compiler handles a large number of variables including global variables that are mostly not related. Thus a sparse representation of constraints is used. It reduces memory storage without impacting the execution time.

3.8 Parsers

PIPS has its own Fortran 77 parser. It has been extended to support HPF and OPENMP. In 2003, the C89 parser from the Berkeley CCur project has been added. It was recently extended to deal with C99 and `#pragma`. The GCC Fortran 2003 parser has been grafted on PIPS and more recent Fortran dialect support is improving.

We have also developed a Scilab/Matlab to C compiler with type inference to parallelize such languages in PAR4ALL.

3.9 Prettyprinters and Runtime Backends

PIPS is a source-to-source compiler and logically can regenerate source code, but as shown in § 2 it can do more. For the classical source generation, C and Fortran prettyprinters are available, with different flavours according to the target: C89 or C99, Fortran 77, 95 or 2003, with or without parallel decorations (OPENMP, various parallel extensions...).

To generate code closer to the assembly for SIMD vector extensions in generic processors or various specific processor instructions, intrinsics functions are used in the internal representation and are generated as-is by the prettyprinter.

To generate language extensions for various accelerators such as CUDA, OPENCL or very specific accelerator languages, we use some post-processing on the prettyprinter output coupled with the PAR4ALL Accel runtime based on C, CPP and C++ to target CUDA and OPENCL. A nice side effect is that we can have OPENMP emulation code almost for free. It is quite useful for debugging since

for example we can use Valgrind on the emulated code with CUDA-like communications. It can also be used by end-users wanting to program directly in low level tools such as CUDA and OPENCL in a portable and less verbose way.

3.10 Internal Representation

The PIPS internal representation is a hierarchical control flow graph (HCFG) with a compact representation and a good trade-off between simplicity and source representation precision. It is common to Fortran and C, easing for example the generation of CUDA or OPENCL directly from Fortran. Semantics information such as polyhedral representations are added as decoration resources. The representation is expressed in the NewGen [25] domain-specific language that generates all the basic object methods (accessors, visitor pattern, constructors/destructors) allowing some elegant programming even though PIPS is mainly written in C [11].

3.11 Consistency and Persistence Manager

Since there are many passes and resources in a compiler, chaining them and providing up-to-date information is quite challenging. This issue is solved with a consistency manager using an *à la* `make` description of dependence relations between resources though passes or analyses. When asking for a resource or explicitly applying a transformation, all the necessary passes are applied in a lazy manner toward constructing the goal asked by the user. Since the dependence relations include the caller and callee notions, this deals also with interprocedural analysis in an elegant recursive way. A persistence manager automatically allows PIPS to be stopped and restarted later, allowing by side effect remote invocation or iterative compilation with try/undo almost for free.

3.12 Pass Managers

All passes and analyses of PIPS are defined to compiler developers through the pass managers. PIPS actually has two of them: the legacy one, `tpips`, uses a shell-like syntax with hard-coded functionalities; the newer exposes a PYTHON API, PyPS, where modules, loops and compilation units are exposed as first-class entities. Using the expressiveness of the PYTHON language, it allows to efficiently compose transformations using complex patterns and to quickly build new compilers. Going a step forward, compilers built this way can be combined into other compilers. E.g the multimedia instruction compiler can be combined with the OPENMP compiler to target multicores with SIMD instruction units. The PAR4ALL automatic CUDA code generator is based on this interface.

4. Code Transformations for Heterogeneous Computing

How does a developer port an application on a GPGPU using CUDA? First, he profiles the application to identify hotspots, generally loops. Then he verifies that the outermost loop is parallel, and transforms the loop body into a CUDA kernel. Finally, he adds the data transfers and the kernel call on the host side. The following subsections briefly describe transformations to automate this process. The goal is not to present each of them in depth. Instead, we show that PIPS analyses provide appropriate abstractions to efficiently implement them.

4.1 Computation Intensity Estimation

Offloading a loop on a GPU is relevant only if the data transfer vs. computational intensity trade-off is clearly in favor of the latter. PIPS provides two relevant analyses to make this decision: execution time estimation (see § 3.5) and convex array region analyses (see § 3.4). The former, when available, is expressed as a polynomial in the program variables, with values in the current memory store. The volume of the array regions is a polynomial and a good estimation of the memory footprint. The asymptotic comparison of these two polynomials, done either at compile time or at run time or a combination of both, makes it possible to decide whether offloading is relevant.

4.2 Outlining

Code fragments marked by the previous phase as computationally intensive can be offloaded to the accelerator. Following the *load work store* idiom, it is necessary to extract these fragments into new functions to represent the *work*. Because the accelerator generally does not use the same input language as the host, this separation also distinguishes host operations from the ones of the accelerator based on their function name. Moreover, the outlined fragment is located in an independent compilation unit. This task is performed in a similar manner as [28] based on the private variables analysis and the *def/use* information from the summary effects (see § 3.1).

4.3 Statement Isolation

Statement isolation is the process of isolating all the data accessed by a statement in newly allocated memory areas. This newly allocated areas simulate the remote memory, but are still representable in the internal representation. PIPS offers a convenient analysis to generate the data copy between the remote memory and the local one: *convex array regions* (see § 3.4). Because usual DMAs can only transfer efficiently rectangular areas, those regions are first over-estimated using their rectangular hull. Then *read regions* are translated into a sequence of host-to-accelerator data transfers and *written regions* are converted into accelerator-to-host data transfers. Additionally the array regions that *may* be written are also copied-in to ensure global consistency. New array variables are allocated to represent the variables in the kernel and receive the copied data. Their dimension is the result of the union of the *read* and *write* regions above and may be lower than the original ones. They are substituted to the old ones in the kernel code.

4.4 Rectangular Symbolic Tiling and Memory Footprint

Thanks to convex array regions, it is possible to compute an over-approximation of the memory footprint of a statement. If it exceeds the accelerator size, the computation cannot be run by the accelerator in a single pass. Assuming the outer statement is a loop nest, it is however still possible to tile its iteration space and to perform the computation in multiple passes. Because the tiling parameters depend on the memory footprint of the inner loops, the tiling is first performed symbolically. Then the memory footprint is computed as a function of these parameters, which leads to a set of inequalities. This set can be augmented by additional constraints such as the number of processing elements which limits the number of iteration of one of the loops. Again, it may not be possible to deduce statically a value for the parameters from these inequalities, in which case decision is postponed until run time.

4.5 Iteration Clamping

When a loop is scheduled on a GPGPU, iteration clamping must be used to ensure only the relevant loop iterations are executed. PIPS makes it possible to guard the execution of the kernel by its preconditions, which ensures no extra iteration is performed.

5. Conclusion

PIPS is a source-to-source transformation framework that contains approximately 300 passes and has deep foundations in abstract interpretation based on linear algebra and the polyhedral model. Despite its age, (it is the project that first used polyhedral methods in compilation in the 80's), the PIPS project has made good fundamental design choices from the beginning that have allowed us to

evolve through subsequent generations with the integration of new tools and features.

The linear algebra framework used in PIPS has a sparse representation, allowing us to represent whole programs in a memory-efficient way and to reason across procedures instead of only over loop nests with affine bounds.

PIPS has been used on a wide range of applications, showing the power of a linear algebra framework coupled with a source-to-source infrastructure. The source-to-source approach allows also to debug the application and the compilation techniques after each transformation pass

PIPS is currently targeting the major issues in compilation today arising from the development of both parallelism (manycores, cloud computing) and heterogeneous computing with strong energy constraints. These trends translate into a need for automatic parallelization with data and communication mapping, communication optimization, and increased memory locality. We showed in this paper how the implementation of these ideas can be readily performed using PIPS as a development infrastructure for application- or architecture-specific (here a GPU) compiler phases.

PIPS is also used in the PAR4ALL commercial product that automatically parallelizes C, Fortran, Scilab and Matlab to OPENMP, CUDA, OPENCL and various accelerators in order to shorten the time-to-market of customer applications. Performance results are shown for real programs in [22].

Acknowledgement

This work is currently funded by the European ARTEMIS SCALOPES and SMECY projects, the French NSF (ANR) FREIA and MédiaGPU projects, the French Images and Networks research cluster TransMedi@ project, and the French System@TIC research cluster OpenGPU project. The authors want to thanks their colleagues working on/with PIPS and PAR4ALL at various places and the so many PIPS contributors that made it possible. We thank Janice ONANIAN MCMAHON for her proofreading.

References

[1] C. Ancourt and F. Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers (CPC'92)*, Vienna.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, 1991.

[3] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static High Performance Fortran code distribution. *Scientific Programming*, 6(1):3–27, 1997.

[4] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electr. Notes Theor. Comput. Sci.*, 267(1), 2010.

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. ICS '08, 2008.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13*, Sept. 2004.

[7] J. Cambonie, S. Guérin, R. Keryell, L. Lagadec, B. Pottier, O. Sentieys, B. Weber, and S. Yazdani. Compiler and system techniques for SoC distributed reconfigurable accelerators. In *SAMOS IV*, July 2004.

[8] F. Coelho. Compilation of I/O communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[9] F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2), Nov. 1996.

[10] F. Coelho and F. Irigoin. Compiling for a heterogeneous vector image processor. In *Workhop on Optimizations for DSP and Embedded Systems (ODES'9)*, Chamonix, France, Apr. 2011.

[11] F. Coelho, P. Jouvelot, F. Irigoin, and C. Ancourt. Data and Process Abstraction in PIPS Internal Representation. In *Workshop on Internal Representations (WIR)*, Chamonix, France, Apr. 2011.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. POPL '78, 1978.

[13] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[14] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In *Languages and Compilers for Parallel Computing*, 1997.

[15] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3), 2000.

[16] F. Devin, P. Boulet, J.-L. Dekeyser, and P. Marquet. Gaspard: A visual parallel programming environment. PARELEC '02, 2002.

[17] P. Feautrier. Toward automatic partitioning of arrays on distributed memory computers. ICS '93.

[18] S. Guelton, F. Irigoin, R. Keryell, and F. Perrin. SAC : An Efficient Retargetable Source-to-Source Compiler for Multimedia Instruction Sets. Technical Report CRI/A-429, Mines Paris-Tech, 2010.

[19] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 1997.

[20] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, 1996.

[21] F. Hannig, H. Dutta, A. Kupriyanov, J. Teich, R. Schaffer, S. Siegel, R. Merker, R. Keryell, B. Pottier, and O. Sentieys. Co-design of massively parallel embedded processor architectures. In *ReCoSoC'05*.

[22] HPC Project. Par4All initiative for automatic parallelization, 2010. `http://www.par4all.org`.

[23] F. Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools For Parallel Scientific Computing*. CNRS-NSF, Sept. 1992.

[24] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. ICS '91.

[25] P. Jouvelot and R. Triolet. NewGen: A Language-Independent Program Generator. Technical Report CRI/A-191, Mines ParisTech, 1989.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[27] E. Lenormand and G. Édelin. An industrial perspective: A pragmatic high end signal processing design environment at THALES. In *SAMOS*, 2003.

[28] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*, Oct. 2009.

[29] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. From OpenMP to MPI : first experiments of the STEP source-to-source transformation. In *ParCO - PARMA*, Sept. 2009.

[30] T. V. N. Nguyen and F. Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27, May 2005.

[31] S. J. Orfanidis. *Introduction to signal processing*. 1995.

[32] L.-N. Pouchet, C. Bastoul, and U. Bondhugula. PoCC: the Polyhedral Compiler Collection, 2010. `http://pocc.sf.net`.

[33] 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)*.

[34] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statement. In *SIGPLAN '86 Symposium on Compiler Construction*.

[35] Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations: extended version. *International Journal of Parallel Programming*, 23, Aug. 1995.

[36] L. Zhou. Complexity estimation in the PIPS parallel programming environment. In *Parallel Processing: CONPAR 92—VAPP V*, 1992.