Polly - Polyhedral optimization in LLVM

Tobias Grosser
Universität Passau
Ohio State University
grosser@fim.uni-passau.de

Hongbin Zheng
Sun Yat-Sen University
st04zhhb@mail2.sysu.edu.cn

Raghesh Aloor
Indian Institute of Technology Madras
raghesh@cse.iitm.ac.in

Andreas Simbürger
Universität Passau
andreas.simbuerger@uni-passau.de

Armin Größlinger
Universität Passau
armin.groesslinger@uni-passau.de

Louis-Noël Pouchet
Ohio State University
pouchet@cse.ohio-state.edu

ABSTRACT

Various powerful polyhedral techniques exist to optimize computation intensive programs effectively. Applying these techniques on any non-trivial program is still surprisingly difficult and often not as effective as expected. Most polyhedral tools are limited to a specific programming language. Even for this language, relevant code needs to match specific syntax that rarely appears in existing code. It is therefore hard or even impossible to process existing programs automatically. In addition, most tools target C or OpenCL code, which prevents effective communication with compiler internal optimizers. As a result target architecture specific optimizations are either little effective or not approached at all.

In this paper we present Polly, a project to enable polyhedral optimizations in LLVM. Polly automatically detects and transforms relevant program parts in a language-independent and syntactically transparent way. Therefore, it supports programs written in most common programming languages and constructs like C++ iterators, goto based loops and pointer arithmetic. Internally it provides a state-of-the-art polyhedral library with full support for $\mathbb{Z}$-polyhedra, advanced data dependency analysis and support for external optimizers. Polly includes integrated SIMD and OpenMP code generation. Through LLVM, machine code for CPUs and GPU accelerators, C source code and even hardware descriptions can be targeted.

Keywords

Loop Transformation, OpenMP, Polyhedral Model, SIMD, Tiling, Vectorization

1. INTRODUCTION

Today, effective polyhedral techniques exist to optimize computation intensive programs. Advanced data-locality optimizations are available to accelerate sequential programs [6]. Effective methods to expose SIMD and thread-level parallelism were developed and are used to offload calculations to accelerators [3, 2]. Polyhedral techniques are even used to synthesize high-performance hardware [15].

Yet, the use of programming-language-specific techniques significantly limits their impact. Most polyhedral tools use a basic, language specific front end to extract relevant code regions. This often requires the source code to be in a canonical form, disallowing any pointer arithmetic or higher level language constructs like C++ iterators and prevents the optimization of programs written in languages like Java or Haskell. Nevertheless, even tools that limit themselves to a restricted subset of C may apply incorrect transformations, as the effects of implicit type casts, integer wrapping or aliasing are mostly ignored. To ensure correctness manual annotation of code that is regarded safe to optimize is often required. This prevents automatic transformations and consequently reduces the impact of existing tools.

In addition, significant optimization opportunities are missed by targeting a programming language like C and subsequently passing it to a compiler. Effective interaction between polyhedral tools and compiler internal optimizations are prevented. The only possible way to pass information are source code annotations like C pragmas. As influencing performance related decisions of the compiler is difficult, the resulting program often suffers from poor register allocation, missed SIMDization or similar problems.

The low level virtual machine (LLVM) [13] is a set of tools and libraries to build a compiler. Constructed around a language and platform-independent intermediate representation (IR) it provides state-of-the-art analyses, optimizations and target code generation. Besides production quality support for x86-32, x86-64 and ARM, there is target code generation for the C programming language, various GPU accelerators or even hardware descriptions. There exist LLVM based static and just-in-time compilers for nine of the ten most used programming languages [14]. Furthermore, there are OpenCL compilers developed by Apple, AMD and NVIDIA, as well as various graphics shader compilers. Due to the modular structure, the human readable IR and a set of helpful tools, development with LLVM is easy and productive.

With Polly we are developing a state-of-the-art polyhedral infrastructure for LLVM, that supports fully automatic transformation of existing programs. Polly detects and extracts relevant code regions without any human interaction. Since Polly accepts LLVM-IR as input, it is programming language independent and transparently supports constructs like C++ iterators, pointer arithmetic or goto based loops.

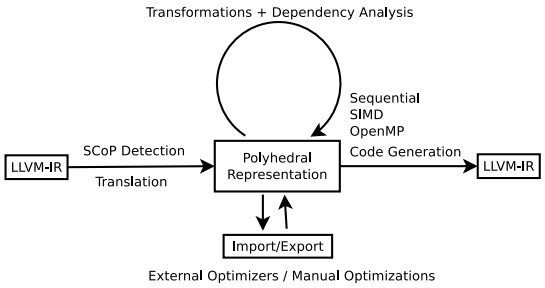

Figure 1: Architecture of Polly

It is built around an advanced polyhedral library with full support for existentially quantified variables and includes a state-of-the-art dependency analysis. Due to a simple file interface it is easily possible to apply transformations manually or to use an external optimizer. We use this interface to integrate Pluto [6], a modern data locality optimizer and parallelizer. Thanks to integrated SIMD and OpenMP code generation, Polly automatically takes advantage of existing and newly exposed parallelism.

In this paper we will focus on concepts of Polly we believe are new or little discussed in the polyhedral community.

2. IMPLEMENTATION

Polly is designed as a set of compiler internal analysis and optimization passes. They can be divided into front end, middle end and back end passes. The front end translates from LLVM-IR into a polyhedral representation, the middle end transforms and optimizes this representation and the back end translates it back to LLVM-IR. In addition, there exist preparing passes to increase the amount of analyzable code as well as passes to export and reimport the polyhedral representation. Figure 1 illustrates the overall architecture.

To optimize a program manually three steps are performed. First of all the program is translated to LLVM-IR. Afterwards Polly is called to optimize LLVM-IR and finally, target code is generated. The LLVM-IR representation of a program can be obtained from language-specific LLVM based compilers. clang is a good choice for C/C++/Objective-C, DragonEgg for FORTRAN and ADA, OpenJDP or VMKit for Java VM based languages, unladen-swallow for Python and GHC for Haskell. Polly also provides a drop in replacement for `gcc` that is called `pollycc`.

2.1 LLVM-IR to Polyhedral Model

To apply polyhedral optimizations on a program, the first step that needs to be taken is to find relevant code sections and create a polyhedral description for them. The code sections that will be optimized by Polly are static control parts (SCoPs), the classical domain of polyhedral optimizations. Extending the polyhedral model and therefore Polly to more general programs is possible as shown by Benabderrahmane [5].

2.1.1 Region-based SCoP detection

Polly implements a structured, region-based approach to detect the SCoPs available in a function. It uses a refined version of the program structure tree described by Johnson [12].

```
for (i = 0; i < n + m; i++)
    A[i] = i;
```

Figure 2: A valid syntactic SCoP. Not always a valid semantic SCoP

A *region* is a subgraph of the control flow graph (CFG) that is connected to the remaining graph by only two edges, an entry edge and an exit edge. Viewed as a unit it does not change control flow. Hence, it can be modeled as a simple function call, which can easily be replaced with a call to an optimized version of the function. A *canonical region* is a region that cannot be constructed by merging two adjacent smaller regions. A region *contains* another region if the nodes of one region are a subset of the nodes of the other region. A tree is called *region tree*, if the nodes of it are canonical regions and the edges are defined by the contains relation.

To find the SCoPs in a function we look for the maximal regions that are valid SCoPs. Starting from the outermost region, we look for canonical regions in the region tree that are valid SCoPs. In case the outermost region is a valid SCoP, we store it. Otherwise, we check each child. After analyzing the tree, we have a set of maximal canonical regions that form valid SCoPs. These regions are now combined to larger non-canonical regions such that, finally, the maximal non-canonical regions that form valid SCoPs are found.

2.1.2 Semantic SCoPs

In contrast to prevalent approaches based on the abstract syntax tree (AST), Polly does not require a SCoP to match any specific syntactic structure. Instead, it analyzes the semantics of a SCoP. We call SCoPs that are detected based on semantic criteria *semantic SCoPs*.

A common approach to detect a SCoP is to analyze an AST representation of the program, that is close to the programming language it is implemented in. In this AST control flow structures like for loops and conditions are detected. Then it is checked if they form a SCoP. Common restrictions that need to be met are the following. There exists a single induction variable for a loop that is incremented from a lower bound to an upper bound by a stride of one. Upper and lower bounds need to be expressions that are affine in parameters and surrounding loop induction variables, where a parameter is any integer variable that is not modified inside the SCoP. The only valid statements are assignments that store the result of an expression to an array element. The expression itself uses side effect free operators with induction variables, parameters or array elements as operands. Array subscripts are affine expressions in induction variables and parameters. There are various ways to extend this definition of a SCoP, which we did not include in this basic definition.

The detection of SCoPs as shown in Figure 2 with an AST based approach is easily possible, however as soon as programs become more complex and less canonical difficulties arise. The AST of a modern language is often very expressive, such that there exist numerous ways a program can be represented. Sometimes different representations can be canonicalized. However, as soon as goto based loops should

be detected, various induction variables exist or expressions are spread all over the program, sophisticated analyses are required to check if a program section is a SCoP. Further difficulties arise through the large amount of implicit knowledge that is needed to understand a programming language. A simple, often overlooked problem is integer wrapping. Assuming `n` and `m` are unsigned integers of 32 bit width, it is possible that $n + m < n$ holds, because of the wrapping semantics of C integer arithmetic [11]. The upper bound in the source code must therefore be represented as $n+m \bmod 2^{32}$, but no polyhedral tool we know of models the loop bound in this way. Further problems can be caused by preprocessor macros, aliasing or C++ (operator) overloading. We believe even standard C99 is too complex to effectively detect SCoPs in it. Tools like PoCC evade this problem by requiring valid SCoPs to be explicitly annotated in the source code. However, this prevents any automatic optimization and significantly limits the impact of polyhedral techniques.

Fortunately, after lowering programs to LLVM-IR the complexity is highly reduced and constructs like implicit type casts become explicit. Furthermore, it is possible to run a set of LLVM optimization passes, that further canonicalize the code. As a result, an analysis that detects SCoPs based on their semantics is possible. LLVM-IR is a very low-level representation of a program, which does not have loops, but jumps and gotos and has no arrays or affine expressions, but pointer arithmetic and three address form operations. From this representation all necessary information is recomputed using advanced compiler internal analyses available in LLVM. Simple analyses used are loop detection or dominance information to verify a SCoP contains only structured control flow. More sophisticated ones check for aliasing or provide information about side effects of function calls.

An especially important aspect is how expressions for the loop bounds, conditions or array subscripts are recovered. On LLVM-IR these expressions are split into individual operations, may be partially hoisted out of loops, or are even further optimized. Optimizations that transform multiplications into shifts or a sequence of additions are very common. As a result, understanding the effects of calculations on LLVM-IR is difficult. Fortunately, LLVM provides an analysis called scalar evolution [8], which calculates closed form expressions for all scalar integer variables in a program. It abstracts away all intermediate calculations including calculations that accumulate values over several loop iterations and describes the value of each scalar as a chain of recurrence [1]. The chain of recurrence depends only on variables with values unknown at compile time and variables describing the loop iteration at which it should be evaluated. As chains of recurrence are more expressive than affine expressions, the work left for Polly is to check if a chain of recurrence describes an affine expression and, if this is the case, translate it into the corresponding affine expression. This recovered affine expression can then be used to represent loop bounds, for example. The same concept is used to recover array accesses from plain memory loads and stores, because address arithmetic is integer arithmetic.

As Polly successfully recovers all necessary information from a low-level representation, there are no restrictions on the syntactic structure of the program source code. A code section is accepted as soon as the LLVM analyses can prove that it has the semantics of a SCoP. As a result, arbitrary control flow structures are valid if they can be written as a well-structured set of for-loops and if-conditions with affine expressions in lower and upper bounds and in the operands of the comparisons. Furthermore, any set of memory accesses is allowed as long as they behave like array accesses with affine subscripts. A loop written with `do..while` instead of `for` or fancy pointer arithmetic can easily be part of a valid SCoP. To illustrate this we show two examples of valid SCoPs in Figure 3.

```
// SCoP with do..while loop
int i = 0;
do {
    int b = 2 * i;
    int c = b * 3 + 5 * i;
    A[c] = i;
    i += 2;
} while (i < N);

// SCoP with pointer arithmetic
int A[1024];
int *B = A;
while (B < &A[1024]) {
    *B = 1;
    ++B;
}
```

Figure 3: Valid semantic SCoPs

2.2 Polyhedral Model

2.2.1 The integer set library

Polly uses isl, an integer set library developed by Verdoolaege [17]. Isl natively supports existentially quantified variables in all its data structures; therefore, Polly also supports them throughout the whole transformation. This enables Polly to use accurate operations on $\mathbb{Z}$-polyhedra instead of using polyhedra in the rationals as approximations of integer sets. Native support of $\mathbb{Z}$-polyhedra simplified many internal calculations and we expect it to be especially useful to represent the modulo semantics of integer wrapping and type casts.

2.2.2 Composable polyhedral transformations

Polly uses the classical polyhedral description [9] that describes a SCoP as a set of statements each defined by a domain, a schedule and a set of memory accesses. The domain of a statement is the set of values the induction variables surrounding the statement enumerate. For example, the statement in Figure 2 has domain $\{[i] : 0 \leq i \leq n + m \bmod 2^{32}\}$. As the statement only has one memory access A[i], the set of memory references for array A is the same. A schedule is a relation which, when applied to the domain, yields the execution times of the statement's operations. In the example the iterations of the loop on i are independent and can be executed in parallel. Therefore, assigning execution time 0 to all iterations by $\{[i] \to [0]\}$ is a valid schedule. By using isl, domains, schedules and memory access sets can be $\mathbb{Z}$-polyhedra in Polly.

In contrast to most existing tools the domain of a statement cannot be changed in Polly. All transformations need to be applied on the schedule. There are two reasons for this. First, we believe it is conceptually the cleanest approach to use the domain to define the set of different statement instances that will be executed and to use the schedule for defining their execution times. As the set of different statement instances never changes there is no need to change the domain. The second reason is to obtain compositionality of transformations. As transformations on SCoPs are described by schedules only, the composition of transformations is simply the composition of the relations representing the schedules.

Applying transformations only on the schedule is simple for transformations like loop fusion, loop fission or loop interchange. Traditionally, strip mining or loop blocking have been transformations that required modifications of the domain, because in the traditional model schedules are affine functions, not relations. But with the relational representation for schedules, these transformations and, we believe, all other desirable transformations can be described by schedules only. For example, strip mining one dimension by four can be expressed by $\{[i] \rightarrow [o, i] : \exists e : 4e = o \wedge o \leq i \leq 3+o\}$.

2.2.3 Export/Import

Polly supports the export and reimport of the polyhedral description. By importing an updated description with changed schedules a program can be transformed easily. To prevent invalid optimizations Polly automatically verifies newly imported schedules. Currently Polly supports the Scoplib exchange format, which is used by PoCC[1] and Pluto [6]. Unfortunately, the Scoplib exchange format is not expressive enough to store information on existentially quantified variables, schedules that include inequalities or memory accesses that touch more than one element. Therefore, we have introduced a simple JSON[7] and isl based exchange format to experiment with those possibilities.

2.3 Polyhedral Model to LLVM-IR

Polly uses CLooG [4] to translate the polyhedral representation back into a generic AST. This AST is then translated into LLVM-IR based loops, conditions and expressions.

2.3.1 Detecting parallel loops

Polly can detect parallel loops automatically and generates, if requested, thread-level parallel code by inserting calls to the GNU OpenMP runtime. This is targeted to automatically take advantage of parallelism present in the original code or exposed by previously run optimizers. To ensure correctness of generated code Polly does not rely on any information provided by external optimizers, but independently detects parallel loops. We present a novel approach how to detect them.

A common approach[2] to detect parallelism is to check before code generation, if a certain dimension of the iteration space is carrying dependences. In case it does not, the dimension is parallel. This approach can only detect fully parallel dimensions. However, during the generation of the generic AST, CLooG may split loops such that a single dimension is enumerated by several loops. This may happen automatically, when CLooG optimizes the control flow, or an optimizers may on purpose generate a schedule that enforces the splitting. With the classical approach either all split loops are detected as parallel or no parallelism is detected at all.

The approach taken in Polly detects parallelism after generating the generic AST and calculates for each generated for-loop individually if it can be executed in parallel. This is achieved by limiting the normal parallelism check to the subset of the iteration space enumerated by the loop. To obtain this subset we implemented an interface to directly retrieve it from CLooG. As a result, we do not need to parse the AST to obtain it. With this enhanced parallelism check parallel loops in a partial parallel dimension can be executed in parallel, even though there remain some sequential loops. This increases the amount of parallel loops that can be detected in unoptimized code and removes the need for optimizers to place parallel and sequential loops in different dimensions.

Polly automatically checks all generated loops and introduces OpenMP parallelism for the outermost parallel loops. By default it assumes parallel execution is beneficial. Optimizers that can derive that for some loops sequential execution is faster may provide hints to prevent generation of OpenMP code. Polly could incorporate such hints during code generation easily, as they do not infect the correctness of the generated code.

2.3.2 Trivially SIMDizable loops

In Polly we also introduced the concept of trivially SIMDizable loops. A trivially SIMDizable loop is a loop that is parallel, does not have any control flow statements in its body and has a number of iterations that is in the order of magnitude of the SIMD vector width. If Polly can find such a loop on the generic AST the loop is not translated into a loop structure, but into a set of vector instructions with a width corresponding to the number of loop iterations.

Polyhedral transformations can now easily introduce SIMD vector code by applying transformations that expose such trivially SIMDizable loops. There is no need to explicitly issue SIMD intrinsics an more, as Polly automatically issues the correct SIMD instructions.

3. EXPERIMENTAL RESULTS

3.1 A hand-optimized example

As an example on how to optimize a SCoP manually we look at a difficult version of matrix multiplication as shown in Figure 4. Each memory access has stride one with respect to a different loop dimension. Therefore, finding an effective way to SIMDize this code is difficult. By importing a new schedule, but not changing anything else, we are able to transform the SCoP into code equivalent to the vectorized one in Figure 4. Polly detects the trivially SIMDizable inner loop and introduces SIMD instructions automatically to execute it.

Figure 5 shows the benefits of the single transformations that Polly applies and compares it to the run times of a clang, GCC and ICC compiled binary. The first run of Polly with

[1] `http://pocc.sf.net`
[2] used for example in Pluto

```
// Plain version
for (k=0; k < 32; k++)
  for (j=0; j < 32; j++)
    for (i=0; i < 32; i++)
      C[i][j] += A[k][i] * B[j][k];


// Vectorized version
for (k=0; k < 32; k++)
  for (j=0; j < 32; j+=4)
    for (i=0; i < 32; i++)
      C[i][j:j+3] += A[k][i] * B[j:j+3][k];
```

Figure 4: Matrix multiplication kernel where each memory access has unit stride with respect to another loop dimension

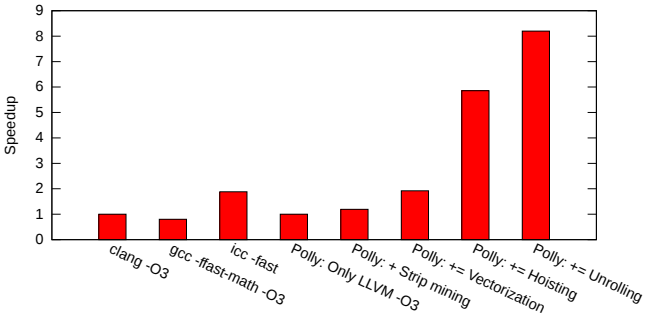

Figure 5: Optimizing non-matching (32x32 float) matrix multiplication with Polly, clang 2.8, GCC 4.5 and ICC 11.1 on Intel® Core™ i5 CPU M 520 @ 2.40GHz

only LLVM `-O3` shows no change to the execution time of clang. This is expected, as the only difference is the translation from LLVM-IR to the polyhedral model and back, which obviously should not introduce any overhead. The next step is the strip mining of the j-loop that creates a trivially vectorizable innermost loop with four iterations. This is where the new schedule is imported and, even without SIMDization enabled, this yields a small runtime improvement. When Polly SIMDizes the newly created trivially vectorizable loop the performance of the ICC compiled binary is reached. The next huge performance increase is obtained from hoisting the loads from B out of the innermost loop. LLVM will do this automatically, if Polly proves that the i-loop is executed at least once. By increasing the LLVM unroll factors runtime is further improved, such that finally an 8x the original performance is reached.

3.2 PolyBench optimized with Pluto

Another interesting experiment is an automatic run of Polly on PolyBench[3], a set of computation intensive programs often used in the polyhedral community. On those benchmarks Polly extracts the relevant SCoPs and optimizes them automatically. We compare clang to clang plus Polly without any optimizations and to clang plus Polly with Pluto based tiling, but still without vectorization and OpenMP parallelism. For 29 out of 30 benchmarks the runtime of the programs compiled with Polly, but without any transformations, is very close to the runtime without Polly. This shows that there is usually no overhead introduced by the translation to the polyhedral model; why the performance degrades for reg_detect slightly is currently under investigation. For one case the runtime is slightly improved, as Polly seems to expose further optimization opportunities to LLVM. As soon as Pluto tiling is applied drastic runtime changes appear. For 9 out of 30 benchmarks Pluto tiling is able to significantly increase the performance, with 6 benchmarks having more than 3x improvements. We consider achieving considerable speedups for a first test run with unoptimized Pluto/Polly interaction quite a success. Improving the Pluto/Polly interaction to exploit the full optimization potential of Pluto is planned future work.

[3] `http://www.cse.ohio-state.edu/~pouchet/software/polybench`

4. RELATED WORK

The work in Polly was inspired by ideas developed in the Graphite project [16], yet Polly uses novel approaches in many areas. For instance, Graphite did not include a structured SCoP detection, even though currently a SCoP detection similar to the one in Polly is developed. Furthermore, Graphite works on the GCC intermediate representation, which is in several areas higher level than LLVM-IR, such that several constructs like multi-dimensional arrays are easily available. Internally Graphite still uses a rational polyhedral library and only in some cases relies on an integer linear programming solver. Furthermore, it does not support existentially quantified variables throughout the polyhedral transformations. Graphite uses the classical parallelization detection before code generation and is not yet closely integrated with the SIMD or OpenMP code generation. In contrast to Polly, it has been tested for several years and is reaching production quality.

The only other compiler with an internal polyhedral optimizer we know of is IBM XL/C. Unfortunately, we could not find any information on how SCoP detection and code generation is done. There exists a variety of source to source transformation tools such as Pluto[4] [6], PoCC or LooPo [10].

5. CONCLUSIONS

Powerful polyhedral techniques are available to optimize programs, but their impact is still limited. The main reasons are difficulties to optimize existing programs automatically. With Polly we are developing an infrastructure that not only is able to optimize C/C++ fully automatically, but also supports a wide range of other programming languages. Based on a state-of-the-art polyhedral library and dependency analysis Polly provides integrated OpenMP and SIMD code generation. We consider it to be an interesting candidate for developing new polyhedral optimizations or increase the impact of existing ones.

Acknowledgements. We would like to thank P. Sadayappan from the Ohio State University, who supported the development of Polly for six months, as well as Albert Cohen, Martin Griebl, Sebastian Pop, and Sven Verdoolaege who largely affected this work. Finally, we would like to thank

[4] `http://pluto.sf.net`

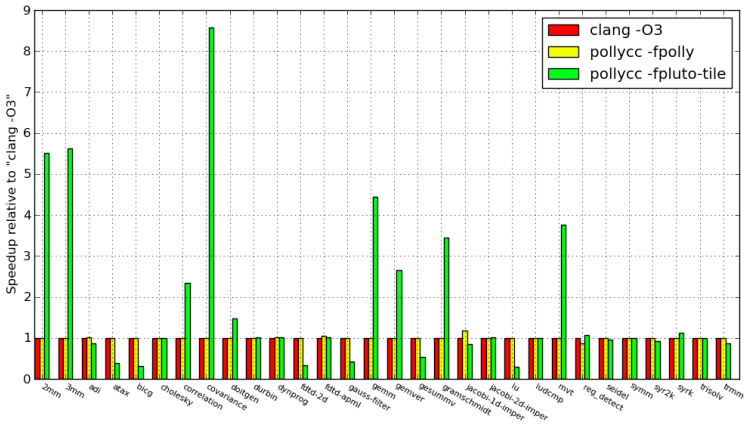

Figure 6: PolyBench 2.0 (with large problem size) optimized with Polly and Pluto tiling compared to clang 2.9rc. Run on Intel® Xeon™ CPU X5670 @ 2.93GHz

Christian Lengauer and Dirk Beyer, who supported Polly with several university projects. This work was funded in part by the U.S. National Science Foundation through awards 0811781 and 0926688.

6. REFERENCES

[1] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '94, pages 242–249, 1994.

[2] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proc. of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, July 2010.

[3] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. 2010.

[4] 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, September 2004.

[5] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303, Mar. 2010.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, 2008.

[7] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[8] R. A. V. Engelen. Efficient symbolic analysis for optimizing compilers. In *In Proceedings of the International Conference on Compiler Construction*, ETAPS CC 'O1, pages 118–132, 2001.

[9] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006.

[10] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In *Languages and Compilers for Parallel Computing*, volume 1239 of *Lecture Notes in Computer Science*, pages 603–604. 1997.

[11] ISO. The ANSI C standard (C99). Technical Report WG14 N1256, ISO/IEC, 1999.

[12] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 171–185, 1994.

[13] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:75, 2004.

[14] TIOBE Software BV. TIOBE programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, Jan. 2011.

[15] T. Risset, S. Derrien, P. Quinton, and S. Rajopadhye. High-Level Synthesis of Loops Using the Polyhedral Model. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis : From Algorithm to Digital Circuit*, pages 215–230. 2008.

[16] 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, Jan. 2010.

[17] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. 2010.