Reconstruction of multi-dimensional arrays in SAPFOR

Nikita Kataev[1][0000-0002-7603-4026], Vladislav Vasilkin[2][0000-0002-7918-1849]

[1] Keldysh Institute of Applied Mathematic RAS Miusskaya sq., 4, 125047, Moscow, Russia
[2] Lomonosov Moscow State University, GSP-1, Leninskie Gory, 11999, Moscow, Russia
kaniandr@gmail.com

Abstract. Low-level representation of programs in the form of LLVM IR allows for various optimizations to improve the quality of program analysis in SAPFOR (System FOR Automated Parallelization). Being the same for different high-level languages, LLVM IR allows us to explore multilingual applications. At the same time, it loses some features of the program, which are available in its higher level representation. One of these features is the multi-dimensional structure of the arrays. Multi-dimensional view improves the accuracy of data dependency analysis, since the linearized representation of arrays with parametric sizes may not be an affine expression and it will not be possible to apply integer linear programming to analyze it. In addition, the use of multi-dimensional arrays allows us to use a multi-dimensional processor matrix and to parallelize a whole loop nests, rather than a single loop in the nest. This way, parallelism of a program is going to be increased. These opportunities are natively supported in the DVM system. This paper discusses the approach used in the SAPFOR system to recover the form of multi-dimensional arrays by their linearized representation in LLVM IR. The proposed approach has been successfully evaluated on various applications.

Keywords: Program Analysis, Semi-automatic Parallelization, SAPFOR, DVM, LLVM.

1 Introduction

The multi-dimensional arrays are widely used in many computational programs. For example, they enable descriptions of the object properties at various points in a multi-dimensional computing space. The DVM system [1, 2] relies on the multi-dimensional view of arrays to increase parallelization opportunity of a program. For example, it proposes specifications which simplify mapping of multi-dimensional arrays to a multi-dimensional grid of processors. Thus, to achieve sustainable performance of parallel programs it is necessary to analyze multi-dimensional data structures in the program.

SAPFOR (System FOR Automated Parallelization) [3, 4] produces a parallel version of a program in a semi-automatic way according to DVMH model of the parallel programming for heterogeneous computational clusters. The DVMH model enables consistent distribution of data between processors of a computational cluster which

are represented as a multi-dimensional grid. The presence of *align* directive in a source code introduces the alignment rules which specify the location of elements of arrays relative to each other. These rules use affine expressions to set a correspondence between points of multi-dimensional index spaces of two objects. To avoid a communication overhead, the elements of different arrays which are used on the same iteration of the loop should be mapped to the same processor. Thus, accesses to arrays in loops and the form of subscript expressions in these accesses imply the alignment rules.

Data dependence analysis is obviously required for program parallelization. The visibility of multi-dimensional view of data allows us to significantly increase the accuracy and reduce the computational complexity of data dependency analysis in many cases. To compute loop-carried data dependencies it is possible to perform the pairwise comparison of expressions which calculate the addresses of accessed elements. If the expressions that calculate the offset relative to the beginning of the array are affine with respect to the induction variables of the enclosing loops, then computation of data dependencies becomes NP-hard problem. In this case integer linear programming (ILP) solvers are applicable. Moreover, to reduce the complexity of the problem being solved, heuristics are used. If subscript expressions in a delinearized accesses satisfy some conditions (for example, expressions are SIV [5] (i.e. contains a single index variable)) the pairwise comparison of subscript expressions drastically reduce the complexity of the analysis. The knowledge of the multi-dimensional view of arrays makes it possible to recover subscript expressions from the linearized representation of array accesses.

It should be noted that a parametric size of an array makes the linearized accesses non-affine so it becomes almost impossible to verify the absence of data dependencies.

The SAPFOR system uses LLVM [6] intermediate representation (IR) to perform program analysis. This representation allows us to analyze programs for diverse programming languages (Fortran, C). So, it becomes possible to overcome multi-lingual issue essential for large-scale computational applications [7]. LLVM also provides the ability to hide program transformation from the user in order to improve the quality of the analysis [8, 9]. Implementation of LLVM-based analysis tool also eliminates the need to consider syntactic features of different programming languages and to analyze the diverse set of available language constructs. To preserve the relation with a higher level representation, the debug information can be used. For these purposes LLVM uses DWARF debugging file format (DWARF [10]) which is common for various languages. However, linearized view of array accesses impedes program analysis. Multi-dimensional view of arrays of known constant size is only visible in LLVM IR. For example, the type `[100 x [200 x float]]` can be used to declare an array of 100 * 200 elements.

There are two purposes of array delinearization in the context of SAPFOR. The first one is the reconstruction of the array view which was multi-dimensional in the source code. And the second one is the search for equivalent the multi-dimensional view for arrays represented in the program source code in a linearized form. In this paper, we are primarily interested in the approach to recover multi-dimensional view

of arrays which were originally multi-dimensional. This follows from the restrictions which DVM system imposes on the programs: the use of multi-dimensional arrays in the source code is required.

2 Delinearization in LLVM

The paper [11] proposes an approach to recovering the multi-dimensional view of arrays of parametric size. The presented approach is partially implemented in the context of LLVM and it was initially focused on the use in the Polly framework [12]. Polly is a high-level loop and data-locality optimizer. It can also exploit loop-level parallelism for systems with shared memory, including GPU. Polly performs IR-level optimization, therefore, the correspondence between delinearized arrays and the source code objects is not required. Loops can be optimized separately, so delinearization is required in the context of the evaluated loop only instead of the entire program. However, consistent delinearization for all accesses in the entire program is required to exploit parallelization opportunity in case of HPC systems with distributed memory.

The paper [11] is focused on recovering the multi-dimensional view of arrays of parametric size only. Separately, Polly implements delinearization for arrays of known constant size. LLVM provides a simple array type to represent sequences of elements in memory. The array types can be nested, so definition of multi-dimensional arrays of known constant sizes is also supported. That means that in this case delinearization is not actually required, since the array type can be used to restore each subscript expression.

If some of array dimensions have parametric sizes while other dimensions have known constant sizes the whole array is treated as an array of parametric size. So, the recovered number of dimensions and recovered multi-dimensional view may differ from the definition of array in the original program.

It also should be noted that to implement delinearization Polly uses its internal data structures which are only partially included in the LLVM core.

Thus, the use of the proposed algorithms in SAPFOR directly becomes impossible.

Let us consider the main points of the approach presented in [11]. The linearized view of the access A [S_0] ... [S_N – 1] to the array A [D_0] ... [D_N – 1] is

$$A + S\_0 * D\_1 *\ldots* D\_N{-}1 + \ldots + S\_N{-}1. \quad (1)$$

In this case, *D_I* is the size of the dimension *I* and *S_I* is the corresponding subscript expression, *I* takes values from *0* to *N – 1*. This equation implies the following ones:

$$C\_N{-}1 * D\_N{-}1 = GCD(S\_0 * D\_1 * \ldots * D\_N{-}1, \ldots, S\_N{-}2 * D\_N{-}1), \quad (2)$$

$$C\_I * D\_I = GCD(S\_0 * D\_1 * \ldots * D\_N{-}1, \ldots S\_I{-}1 * D\_I * \ldots D\_N{-}1) / (D\_I{+}1 *\ldots* D\_N{-}1),\ 0<I<N{-}1. \quad (3)$$

Thus, in order to calculate the size of the dimension I, it is necessary to find the largest common divisor (GCD) of the terms in (1) which are located to the left of the term I, and to divide the GCD into the product of the previously calculated sizes of dimensions from I + 1 to N – 1. The factor C_I is not equal to 1, for example, if the same factor is present in each subscript expression. The GCD function performs a symbolic calculation of the largest common divisor of all its parameters.

The main difficulties of delinearization are as follows:

1. extract the correct number of terms (equal to the dimensionality of the array) from the equation (1) which calculates the address of the array element,
2. arrange these terms in accordance with the order of dimensions,
3. determine the value of the coefficient C_I.

The number of terms in the equation (1) may differ from the number of array dimensions, if some of the subscript expressions are equal to zero, if some of products have computable constant values or if in (1) some of brackets were removed.

The authors of [11] extracts only terms that contain loop induction variables. They ignore constant factors and the coefficient C_I is assumed to be equal to 1. The terms are sorted according to the number of factors. As a result, the delinearized representation of the array may not correspond to the representation of the array in the original program. This is valid since the main purpose of delinearization in Polly is to get affine subscript expressions.

The delinearization in SAPFOR is based on the idea of the algorithm used in Polly, but contains some features that enables the recovering a multi-dimensional view of the array in the correspondence with the original array definition in a source program.

3 Delinearization in SAPFOR

First of all, it is worth noting that we are considering delinearization for arrays that were multi-dimensional in the original program. For the rest of the arrays, a reconstruction of multi-dimensional view may be useful in order to replace accesses to these arrays in a source code with equivalent accesses to multi-dimensional arrays. In this case delinearization does not depend on representation of arrays in a source code, so the approach implemented in LLVM can be used.

As mentioned above, for arrays of known constant size delinearization is not required. Thus, we consider arrays which have at least one dimension with unknown size.

The *getelementptr* LLVM instruction is used to get the address of an element of some aggregate data structure (see Fig. 1).

```
%6 = zext i32 %I.0 to i64
%7 = mul nuw nsw i64 %6, %1
%arrayidx11 = getelementptr inbounds [2 x double], [2 x double]* %vla, i64 %7
%8 = zext i32 %J.0 to i64
%arrayidx13 = getelementptr inbounds [2 x double], [2 x double]* %arrayidx11, i64 %8
%arrayidx14 = getelementptr inbounds [2 x double], [2 x double]* %arrayidx13, i64 0, i64 1
```

Fig. 1. An example of LLVM IR which calculates an address of an element A[I][J][1] of an 3-dimentional array with M * N * 2 size.

The arguments of this instruction are the address of the beginning of the memory location and one or more indices which are used to calculate the offset relative to the specified base address. In the case of multi-dimensional arrays, the indices are terms from (1) which determine an offset according to each array dimension or explicit subscript expressions (without multiplying by sizes of the array dimensions).

Consider the example of calculating the address of an array element in Fig. 1. The first *getelementptr* instruction shifts the address of the beginning of the array `A` (`%vla`) by the value `I*N` (`%7`), the next instruction shifts the received address by the value `J*2(%8)`. It should be noted that multiplication by the size of the last dimension of the array is performed implicitly by the *getelementptr* instruction (register `%8` contains only the value of the variable `J`). This is due to the fact that the size of the last dimension is fixed, and the array `A` in LLVM IR is of the type `[2 x double]*`.

The dependence of the number of arguments of the *getelementptr* instruction on the dimensionality of the array can be considered a heuristic (it is possible to construct the equivalent LLVM IR by replacing all the *getelementptr* instructions in Fig. 1 with two parameters: the address of the array and the previously calculated offset), similar to that used in [11] to highlight terms. This allows us to draw a conclusion about the dimensionality of the array and to determine the number and order of terms in a linearized representation of a corresponding array access.

To investigate the dimensionality we use instructions that obviously refer to individual elements of the array, such as *load* and *store*. Other instructions (for example, function calls) may take as an argument an entire dimension of an array. This means that the number of arguments in the *getelementptr* instruction may be less than the number of array dimensions. Another obstacle is the use of zero subscript expressions. Such expressions will be omitted from the *getelementptr* instruction. To accurately determine the dimensionality of an array, all accesses to its elements are considered and among them, accesses with the maximum number of dimensions are selected.

In many cases the dimensionality is also presented in the debug information. It also contains description of dimensions of the known constant size. Moreover, a variable which specifies size of a dimension in a source code sometimes is also known.

As the next step, we derive the sizes of unknown array dimensions. For this we use equations (2) and (3) mentioned in the previous section. To reduce the error probability, we calculate the largest common divisor of terms obtained for all elementwise accesses to a given array within the analyzed function. We use the shape of *getele-*

mentptr instruction to extract terms so it allows us to process accesses with non-affine and constant subscript expressions. It should be noted that the C_I coefficient is not equal to 1 only if all subscript expressions that correspond to dimensions in the range [0, I-1] are products with the same factor. In this case it is not possible to accurately determine value of C_I, so we assume it is equal to 1. Hence, the size of the corresponding dimension considered a product with C_I factor. Then the correspondence of a recovered multi-dimensional view with array declaration in a source code will be examined later when a source-to-source transformation is performed.

After the size of the array has been calculated, we extract subscript expressions for each access to the array. Starting from the 0 dimension, the terms are symbolically divided by the product of the sizes of the subsequent dimensions. If the number of terms for a given access is less than the dimensionality of the array, and the product of the sizes of the subsequent dimensions does not divide the current term, then we assume the subscript expression is equal to 0. Thus, zero indices omitted in the *getelementptr* instruction are restored. It is important to note that at this step we process all accesses to arrays, including those that could not be used to extract the dimensionality of the array.

Arguments of the *getelementptr* instruction can have a rather complex structure and nested type casts. For the successful computation of the greatest common divisor, as well as for performing symbolic division of terms, it is necessary to simplify expressions and to remove brackets. However, a type casting may change the results of calculations. This means that compilers disallow brackets elimination in many cases to keep precision of computations. This restriction often prevents Polly from realizing delinearization opportunity. In the SAPFOR system, a recovered multi-dimensional view is not used to generate executable code. This view allows us to investigate data dependencies, to build data distribution, and then to insert corresponding DVMH directives into the source code. The DVM system gives a dynamic tool for a functional debugging. So, if necessary it could be used to check correctness of inserted DVMH-directives in the program. In addition, SAPFOR static analyzer implements a command line option -fsafe-type-cast, which prevents analysis and transform passes from unsafe type casting.

4 Evaluation

We used automatically generated test programs to check the capabilities of the described approach and its implementation. In order to cover maximally all possible cases of using arrays, the generated tests were classified by the following parameters:

- the number of dimensions of the array,
- a way in which sizes of dimensions are specified (variables (C99 variable length arrays), enumerators, macros, and literal constants of various integer types),
- memory allocation: dynamic and compile-time arrays,
- scope of an array which can be local (array declared in the function body or passed as a parameter) or global,

- various kinds of subscript expressions (containing constants, loop induction variables and other integer variables)
- the presence of type conversion (explicit and implicit) in array declarations and accesses to arrays.

Each generated test contains a C99 program as well as the expected delinearization result presented in the JSON format. The ability to generate the delinearization result in JSON format was also added to SAPFOR static analyzer.

We use Ctestgen library [13] to automatically generate and run tests, as well as to analyze the results of launches. The library has been developed in Python 3 by one of the authors of this paper. It is distributed under the MIT license and is available on GitHub [13].

Table 1 shows an example of a generator which creates functions that compute the sum of their arguments. An example of a generated program is shown in Table 2.

Table 1. An example which generates functions to compute sum of their arguments.

```
From ctestgen.generator import TestGenerator
class ExampleTestGenerator(TestGenerator):
  def _generate_programs(self):
    generated_programs = list()
    for i in range(2, 6):
      # Create list of arguments (from 0 to i)
      sum_arguments = [Int('num_' + str(arg_idx)) for
arg_idx in range(i)]
      # Declare a function with a specified name
      # and number of arguments. The function will
      # return value of type type int.
      sum_function = Function('sum_' + str(i) + '_nums',
Int, sum_arguments)
      sum_result = Int('sum')
      # Define function body.
      sum_body = CodeBlock(
        Assignment(VarDeclaration(sum_result),
Add(sum_arguments)),
        Return(sum_result))
      sum_function.set_body(sum_body)
      # Create a program with a single function.
      example_program = Program('sum_' + str(i))
      example_program.add_function(sum_function)
      generated_programs.append(example_program)
    return generated_programs
# Generate the entire test suite.
example_generator = ExampleTestGenera-
tor('example_generator_output')
example_generator.run()
```

A generated test program is an object of the *Program* class, which consists of include directives (*Include* class), macro definitions (*Define* class), enumerations (*Enum* class), global variables (*Var* class) and functions (*Function* class). A function definition comprises a name, a list of arguments and a body which is a block of code. To create a generator, you need to inherit the abstract class `ctest-gen.generator.TestGenerator` and to override the `_generate_programs()` method, which returns the abstract syntax trees of the described program in the C programming language.

Table 2. An example of generated program.

```
int sum_3_nums(int num_0, int num_1, int num_2) {
  int sum = num_0 + num_1 + num_2;
  return sum;
}
```

All generated programs are used as input for an application that should be tested (for example, SAPFOR static analyzer). The library determines whether each test is successfully completed and collects statistic of execution of the whole test suite. It is also possible to compare results of a new launch against results of any previous launches. To run tests, you should inherit the abstract class `ctest-gen.runner.TestRunner` and override the `_on_test()` method, which is called for each program in the given directory.

In the context of SAPFOR the delinearization module is used to construct a parallel version of the original sequential program, as well as to analyze program properties (for example, to determine data dependencies). As was noted in [9], in order to improve the quality of the analysis of the program, its preliminary transformation is required in many cases. The approach proposed in [8] allows us to restore original program properties after transformation. So, IR-level transformation hidden from the user becomes possible. The general analysis execution scheme in SAPFOR is as follows: some analysis is performed before the transformation of LLVM IR, and then IR-level transformation is performed, after that, the analysis is repeated and the previously obtained results are refined. Thus, the results of the analysis will be associated with the objects of the original program which is not transformed. The delinearization module is launched at each step of the analysis in order to perform data dependence analysis. To determine data dependencies, the tests described in [5] are used. These tests were already implemented in the context of LLVM. So, we modify a corresponding pass to enable the usage of the devoted delinearization approach.

The implemented delinearization module in conjunctions with data dependence analysis was also manually checked on NAS Parallel Benchmarks 3.3 [14] and Polybench/C the Polyhedral Benchmark suite 4.2.1 [15].

5 Conclusions

In this paper, we propose an approach to reconstruction the multi-dimensional view of arrays in the C99 language, which presented in lower level LLVM representation in a linearized form. The presented approach relies on the debug information available in LLVM and the view of low-level instructions which calculate the offsets relative to the address of the array beginning. LLVM uses common debugging format to address the requirements of diverse programming languages. In the future works, this format allows us to apply the developed approach to recovering multi-dimensional arrays in Fortran and to avoid additional analysis of constructs in higher language.

The discussed approach is based on the idea used in the delinearization module, which is implemented in LLVM and Polly. However, in contrast to it our implementation, it provides a more accurate correspondence between the delinearized array and the original definition of a multi-dimensional array in the original program. This advantage is essential in the context of source-level parallelization which is a primary goal of SAPFOR development.

It may be useful to apply the presented approach along with the approach implemented in LLVM to recover multi-dimensional view of arrays which are explicitly linearized in a source code. We believe that such research will allow us to implement source-to-source program transformation which replaces linearized array accesses with delinearized ones.

The source code for the SAPFOR system is available on GitHub [16].

This work was partially supported by Presidium RAS, program I.26 "Fundamentals of creating algorithms and software for advanced ultra-high performance computing".

References

1. Konovalov, N.A., Krukov, V.A, Mikhajlov, S.N., Pogrebtsov, A.A.: Fortan DVM: a Language for Portable Parallel Program Development. Programming and Computer Software 21(1), pp. 35–38 (1995).
2. Bakhtin, V.A., Klinov, M.S., Kriukov, V.A., Podderiugina, N.V., Pritula, M.N., Sazanov, Iu.L.: Rasshirenie DVM-modeli parallelnogo programmirovaniia dlia klasterov s geterogennymi uzlami. Vestnik Iuzhno-Uralskogo gosudarstvennogo universiteta, seriia "Matematicheskoe modelirovanie i programmirovanie", №18 (277), vypusk 12. Cheliabinsk: Izdatelskii tsentr IuUrGU. S. 87–92 (2012).
3. Klinov, M.S., Kriukov, V.A.: Avtomaticheskoe rasparallelivanie Fortran-programm. Otobrazhenie na klaster. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo, No. 2, S. 128–134 (2009).
4. Bakhtin, V.A., Zhukova, O.F., Kataev, N.A., Kolganov, A.S., Kriukov, V.A., Podderiugina, N.V., Pritula, M.N., Savitskaia, O.A., Smirnov, A.A.: Avtomatizatsiia rasparallelivaniia programmnykh kompleksov. Nauchnyi servis v seti Internet: trudy XVIII Vserossiiskoi nauchnoi konferentsii (19–24 sentiabria 2016 g., g. Novorossiisk). M.: IPM im. M.V. Keldysha, S. 76–85 (2016). https://doi.org/10.20948/abrau-2016-31, last access 05.12.2019.

5. Goff, G., Kennedy, K., Tseng, CW.: Practical Dependence Testing. In: ACM SIGPLAN 1991 Conference on Programming language design and implementation (PLDI ’91), pp. 15–29. ACM, New York, NY, USA (1991).
6. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: 2004 International Symposium on Code Generation and Optimization (CGO’04). Palo Alto, California (2004).
7. Bakhtin, V.A., Zhukova, O.F., Kataev, N.A., Kolganov, A.S., Kriukov, V.A., Kuznetsov, M.Iu., Podderiugina, N.V., Pritula, M.N., Savitskaia, O.A., Smirnov, A.A.: Parallelization of Software Packages. Problems and Prospects. In: CEUR Workshop Proceedings, vol. 2260, pp. 63–72 (2018).
8. Kataev N.A.: Application of the LLVM Compiler Infrastructure to the Program Analysis. In: SAPFOR. Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, vol 965, pp. 487–499. Springer, Cham. (2018) https://doi.org/10.1007/978-3-030-05807-4_41, last access 05.12.2019.
9. Kataev, N.A., Kozyrev, V.I.: Preobrazovanie programm na vysokourovnevom iazyke programmirovaniia na osnove rezultatov analiza nizkourovnevogo predstavleniia programm v sisteme In: SAPFOR. Parallelnye vychislitelnye tekhnologii: XIII mezhdunarodnaia konferentsiia, PaVT'2019, g. Kaliningrad, 2–4 aprelia 2019 g. Korotkie stati i opisaniia plakatov. S. 251–262. Izdatelskii tsentr IuUrGU, Cheliabinsk (2019).
10. Dwarf 3 Standard. http://eagercon.com/dwarf/dwarf3std.htm, last access 05.12.2019.
11. Grosser, T., Pop, S., Ramanujam, J., Sadayappan, P.: On recovering multi-dimensional arrays in Polly. In: 5th International Workshop on Polyhedral Compilation Techniques, IMPACT 2015. pp. 1–9 (2015).
12. Polly - Polyhedral optimizations for LLVM. https://polly.llvm.org/, last access 05.12.2019.
13. Ctestgen. https://github.com/VolandTymim/ctestgen, last access 05.12.2019.
14. Seo, S., Jo, G., Lee, J.: Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In: 2011 IEEE International Symposium on. Workload Characterization (IISWC), pp. 137–148 (2011).
15. PolyBench/C the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench.html, last access 05.12.2019.
16. SAPFOR. https://github.com/dvm-system, last access 05.12.2019.