LLVM Based Parallelization of C Programs for GPU

Nikita Kataev(✉)

Keldysh Institute of Applied Mathematics RAS, Moscow, Russia
kaniandr@gmail.com

Abstract. The paper proposes an approach to semi-automatic program parallelization in SAPFOR (System FOR Automated Parallelization). SAPFOR proposes opportunities to perform user-guided source-to-source program transformations and to reveal implicit parallelism in sequential programs. The LLVM compiler infrastructure is used to examine a program and Clang is used to perform source-to-source program transformation. This paper highlights benefits of IR-level (Intermediate Representation) program analysis which allows us to apply low-level program transformations to investigate properties of the original program. To exploit program parallelism SAPFOR relies on DVMH which is a directive-based programming model. We use subset of C-DVMH language which allows us to run parallel program on GPU as well on multiprocessors. Evaluation of presented approach has been performed using the C version of the NAS Parallel Benchmarks.

Keywords: Program analysis · Program transformation · Semi-automatic parallelization · SAPFOR · DVM · GPU · LLVM

1 Introduction

Today's parallel hardware platforms are usually heterogeneous and they are not only equipped with multi-core processors but also provide accelerators. In order to fully utilize the available resources, the developers have to update existing software which relies on sequential programming models. To start this time-consuming effort the developers choose from a large set of available approaches to parallel programming.

Low-level data parallel programming models (CUDA, OpenCL) allow us to achieve the best performance but at the same time require the greatest effort. Pragma based models (OpenMP, OpenACC, DVMH [1,2]) simplify programming and increase software maintainability while still providing high performance. DSLs [3–5] and corresponding compilers automate the development of high-performance parallel programs in a given domain. And finally, general purpose libraries enable exploiting heterogeneous platforms through the use of standard high-level programming languages.

V. Voevodin and S. Sobolev (Eds.): RuSCDays 2020, CCIS 1331, pp. 436–448, 2020.
https://doi.org/10.1007/978-3-030-64616-5_38

However, any of these diverse approaches still requires expert knowledge. In this situation, the development of user assistance tools can significantly reduce the cost of parallel programming. The most desirable tools are automatic parallelizing compilers which return a fully parallelized source code for a given sequential one [6–8]. Unfortunately, a generated code of such compilers may not be optimal. In this case, successful program optimization may require preliminary manual transformation of a sequential source code or even further manual optimization of a generated one. Other tools are applicable only on some stages of parallelization [9,10]. For example, they assume that parallelism will be exploited in a manual way, while static or dynamic dependence analysis or source code profiling could be done automatically. Some of them make suggestions how to improve performance of already written parallel program.

Alternative solution is to follow an implicit parallel programming methodology [11,12]. This implies that the programmer is aware that the program must be well-formed for automatic parallelization. Thus, he should be able to increase algorithm-level parallelism, still relies on expressivity of standard sequential programming languages. It is also possible to guide the compiler by the hints which emphasis high-level program properties which are essential for parallelization.

This paper is devoted to the System FOR Automated Parallelization (SAPFOR) [13] which combines approaches mentioned above to automate development of parallel programs.

SAPFOR relies on an implicitly parallel programming model. This means that the system includes an automatic parallelizing compiler and it does not require the user to parallel program explicitly. The system implements both static and dynamic analysis techniques which complement each other. Thus, the static analysis reduces overheads of program evaluation at runtime [14]. In general, dynamic analysis tools are input sensitive and the application of static analysis techniques also reduces the number of analysis results that the user must control. SAPFOR implements LLVM [15] based static analysis. The paper [16] shows how some kind of IR-level (Intermediate Representation) transformations are used in SAPFOR to increase the quality of program analysis. It also argues that despite the use of a low-level program transformations, the analysis report is closely related to the original high-level source code.

The system also provides the user with a set of automatically performed source-to-source transformations (inline expansion, dead code elimination, expression propagation and other) that he can apply to the original sequential program. Guided by the analysis report, the programmer may choose some of them for automatic execution.

Unlike traditional compilers SAPFOR performs source-to-source parallelization and produces a parallel version according to high-level DVMH parallel programming model [1,2]. It was designed to create parallel programs of scientific-technical calculations for heterogeneous computational clusters. C-DVMH is a directive based programming language. The programmer can annotate a C source code to highlight regions of code that should be executed in parallel. Thus, application of DVMH model improves parallel program maintainability and the developer also can optimize it if necessary. Using DVMH extremely simplifies the development of an automatic parallelizing compiler since it is not necessary to use various parallel programming models to generate programs for heterogeneous parallel platforms. Moreover, DVMH runtime system controls low-level data transfer and synchronization and it makes some optimizations of data transfer between CPU and accelerators. Hence SAPFOR should not insert low-level specifications of these operations in a source code.

The rest of the paper is organized as follows. Section 2 presents our approach to automatic parallelization of well-formed sequential programs. It also focuses on the application of lower-level transformations to increase the quality of program analysis. Section 3 outlines the implementation details of the interaction of the higher level transform passes, which perform program parallelization, and lower level analysis passes. Section 4 is devoted to semi-automatic parallelization of the C version of the NAS Parallel Benchmarks [20]. It summarizes the necessary source-to-source transformations and presents the performance results. Section 5 discusses the related work and finally Sect. 6 concludes this paper.

2 Automatic Parallelization

In this paper we consider parallelization for compute devices with shared memory. This means that a parallel program can be run on multi-core CPU or accelerator. For this purpose DVMH model requires that three kinds of annotations be inserted into the source code:

– specifications of the loops which can be executed in parallel, as well as specifications of private and reduction variables,
– specification of the compute regions which can be executed on the accelerators, each region may enclose one or more parallel loops,
– high-level specifications of data transfer between a memory of CPU and a memory of accelerator (actualization directives).

All specifications are in the form of directives. Each directive may contain number of clauses. To make sure the insertion of these directives is permissible, the automatic parallelizing compiler must investigate the properties of the code section to be parallelized. There are two main groups of these properties. Firstly, these are the properties of the program variables and corresponding memory locations. Secondly, these are properties associated with the control flow of the program.

The first group of properties includes loop-carried data dependencies, spurious dependencies, input, output and local data for compute regions. The second group includes summary information on control flow of each function and loop. It is necessary to identify whether function calls have side effect (including I/O operations), whether a program may terminate inside a function call, whether a function captures a pointer (i.e. it makes any copies of the pointer that outlive the function call). It is also necessary to make sure that there is no recursion leading to nested compute regions, including due to indirect function calls.

The paper [16] discusses the implementation of static analysis in SAPFOR. It introduces a novel data structure which is called a source-level alias tree. The source-level alias tree depicts the structure of accessed memory and it allows us to apply transform passes to improve the quality of the source program analysis. This means that SAPFOR analyzes transformed program and propagate its properties to the original source code. Moreover, it is possible to make property-sensitive transformations, i.e. to make some transformation to analyze one kind of properties and another transformation to analyze another kind of property.

In that way, the source-level alias tree is suitable to examine the first group of properties which are necessary for automatic parallelization. If transform passes do not cross bounds of the analyzed code section (function or loop) it is safe to investigate the second group of properties.

After the properties of the program have been explored, SAPFOR looks up for sections of code that can be executed in parallel. At this point, the original source code is processed.

In the first step, we use a depth-first ordering [17] to traverse strongly connected components of a call graph. To avoid explicit recursion, strongly connected components with a single function are considered only. As a result, we prevent the appearance of nested compute regions which are prohibited in the DVMH model.

In the next step, the body of the visited function is processed. Depth-first search of a loop tree allows us to find the outermost loops which can be parallelized. For each loop the following constraints are examined:

1. Safety of control flow. That means the absence of function calls which have side effects, the absence of multiple exits outside the loop body, the absence of I/O operations inside the loop body, the absence of indirect calls to user-defined functions.
2. Safety of memory accesses. That means the absence of loop-carried data dependencies and captured pointers. If a pointer references a privatizable variable and if this pointer is captured, then, after variable privatization, the relation between the pointer and the variable will be lost. Spurious dependencies such as private and reduction variables are allowed.
3. Direction of data usage. An input data is intended to have the newest values at the beginning of the loop. An output data is updated in the loop and is live [17] at any exit from the loop. A local data are updated in the loop, but the values of corresponding memory locations are not used outside the loop. This property will be useful for data transfer optimizations at runtime if this loop is supposed to be executed on GPU. The corresponding clauses will follow a compute region which surrounds this loop.
4. Canonical loop form according to the OpenMP [18] standard. DVMH as well as OpenMP disallows parallelization of a loop that does not have canonical loop form. Source-level alias tree is useful to ensure that loop boundaries and step are loop invariant expressions.
5. The ability to express properties of memory locations with DVMH directives. A source-level alias tree allows us to represent any memory location in the

program. However, directive based programming models have some restrictions on variables listed in clauses. For example, it is not possible to privatize memory that is allocated in the heap. The other case is accesses to a global memory inside the loop body. If there is a call which accesses this global memory, the corresponding variable cannot be placed in *private* or *reduction* clauses. Each new item is allocated for the corresponding variable listed in these constructs. Hence these items are not associated with global memory which is accessed in callees.

6. The ability to collapse iteration spaces of nested loops into one larger iteration space. The corresponding specification in the *parallel* directive is similar to collapse clause in OpenMP [18]. It increases the amount of computation, which is especially important for running a program on accelerators.

If all constraints are satisfied, the current loop will be parallelized and corresponding DVMH directives will be created. Some constraints can be relaxed to allow parallel execution on CPU even if the utilization of GPU is not possible. For instance, calculation of data usage is not necessary in this case and data transfer specifications can be omitted in a parallel code.

In the last step, we optimize the placement of data transfer specifications in a source code. We use postorder traversal [17] to prepare data for accelerator as early as possible and to request data from the accelerator as late as possible. The call graph and loop trees are traversed. Neighboring regions at the same level of a loop tree are joined. Even if regions cannot be joined compiler tries to insert actualization directives before the first region and after the last one to avoid data transfer between computations. For loops which are not parallelized we try to move actualization directives outside the loop body and for functions we move data transfer outside the callees.

3 Implementation Details

We have implemented proposed approach to automatic parallelization in SAPFOR. Low-level LLVM IR is used to perform program analysis and source-to-source transformations rely on Clang AST (Abstract Syntax Tree). LLVM 7.1.0 is currently supported.

As stated in the previous section, we need to have a transformed representation of the program in the form of a LLVM IR as well as its original representation. Transformed representation is suitable to analyze memory locations and to build summary information on control flow while the parallelization pass also investigates original LLVM IR. Maintaining the correspondence between original LLVM IR and Clang AST is necessary for source-to-source transformations.

For this purpose, a separate thread (analysis server) can be started inside the automatic compiler (Fig. 1). It clones the original LLVM IR and then it performs analysis and transform passes according to [16].

The client thread, which is responsible for program parallelization, requests the necessary information from the server. A source-level alias tree is used to

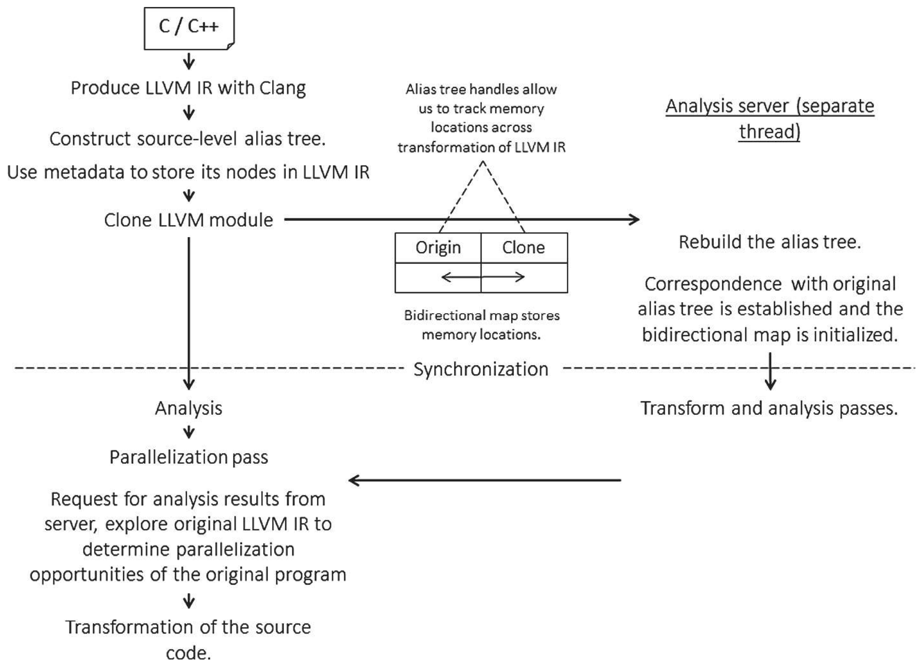
C / C++
Produce LLVM IR with Clang
Construct source-level alias tree.
Use metadata to store its nodes in LLVM IR
Clone LLVM module
Alias tree handles allow us to track memory locations across transformation of LLVM IR
Origin
Clone
Bidirectional map stores memory locations.
Analysis server (separate thread)
Rebuild the alias tree.
Correspondence with original alias tree is established and the bidirectional map is initialized.
Synchronization
Analysis
Parallelization pass
Request for analysis results from server, explore original LLVM IR to determine parallelization opportunities of the original program
Transformation of the source code.
Transform and analysis passes.

Fig. 1. Implementation scheme of the automatic parallelizing compiler in SAPFOR

synchronize the analysis results. An instance of the alias tree will be built both on the client side and on the server side. As noted in [16] for the alias tree, special handlers are implemented that allow the compiler to track memory locations across rebuilding of the alias tree after IR transformation. These handlers are also used to establish correspondence between objects on the server and on the client.

A separate LLVM pass has been created to start analysis server. Built-in support for threads, which is included in C++11, is used. It is possible to create multiple analysis servers at the same time in order to examine transform-sensitive properties. A separate pass is also implemented to transform original program.

4 Evaluation

The applicability of the implemented approach was examined on the C versions of the NAS Parallel Benchmarks (NPB) [19]. In this section we consider in detail the semi-automatic parallelization of three benchmarks: EP (Embarrassingly Parallel), BT (Block Tri-diagonal solver) and CG (Conjugate Gradient). Each of these programs has features that affect the complexity of its analysis and further parallelization. This section highlights SAPFOR capabilities which are helpful to overcome these issues. We also summarize source-to-source program transformations which were applied to improve the quality of the source program analysis.

We performed the evaluation on 6-cores processor Intel Xeon CPU E5-1660 v2, 3.70 GHz with active Hyper Threading (2 threads per core) and with Turbo Boost disabled. GPU experiments were performed on GPU GeForce GTX 1660 Ti. We use Intel Compiler 19.0.2.187 for all tests.

4.1 Semi-automatic Transformation

We made some simple preliminary transformations of benchmarks manually to deal with the limitations of the current version of SAPFOR.

Firstly, each benchmark consists of several files which should be merged together to allow inter-procedural analysis of programs. The ability of Clang to merge together several ASTs does not work well for large programs, especially ones which use the C standard library. On the other hand, LLVM comprises a tool which allows us to obtain an LLVM IR for each file in order to subsequently generate a single LLVM IR for all files. Although this approach is applicable to analyze programs, at the moment, SAPFOR suffers from inability to establish a correspondence between the original higher level sources and a single LLVM IR. Thus, this inability prevents source-to-source transformations and it implies us to merge all sources except header files manually.

Secondly, the presence of macros drastically complicates source-to-source program transformation. For instance, macro with the same name may have different meanings in distinct program regions. Moreover, changes of macro definition after program transformation may lead to changes in the transform region which are unobvious for the user. We replaced definitions of all integer constants with enumerations and definitions of floating point constants with const-qualified variables. However, the wide usage of the preprocessor in real-world applications [21] does not allow us to rely on the absence of macros in programs of scientific-technical calculations as well. In the future versions of SAPFOR the user will be able to force the transformations across presence of macros in the transform region.

The time of the analysis of the merged files, as well as the size of each benchmark in the number of lines of code are given in Table 1. The original versions comprise source files with mentioned above preliminary transformations performed on them. The transformed versions were obtained after manual and user-guided automatic transformations. And finally, the parallel C-DVMH versions were obtained after automatic parallelization.

The increase in code size of the transformed versions is primarily due to the inline expansion. This is the most important transformation which significantly reduce the complexity of program analysis. At this moment SAPFOR implements some kinds of inter-procedural analysis known as classical methods of interprocedural summary dataflow analysis [22]. Unfortunately, as mentioned in [22] this summary information is too coarse to prove the absence of data dependencies. Moreover, in the presence of pointers which are essential for every C program it is necessary to ensure that the callee does not make any copies

Table 1. The analysis time(s) of the NAS Parallel Benchmarks (NPB)

Benchmark	Original		Transformed		Parallel
	Lines	Time (s)	Lines	Time (s)	Lines
BT	3488	46.15	8805	2213.09	8954
CG	1283	0.59	1460	0.67	1515
EP	623	0.2	908	0.4	947

of the pointer that outlive the callee itself. The inline expansion allows SAPFOR to perform pairwise comparison of array accesses to determine whether dependences exist between two subscripted references [23] to this array in the loop nest. In case of inline expansion the loop nest may grow notably causing significant rise of analysis time (up to 37 min for BT).

Call to a function with arguments of pointer type (this is an CG case) is another reason for inline expansion because invocation context should be analyzed to investigate whether or not two pointers can point to the same object. The usage of *restrict* keyword in a source code is another way to overcome this issue.

Along with the choice of transformations, the user can also control the analysis options. As presented in [24] array delinearization is an important technique which is implemented in SAPFOR. It significantly reduces the complexity of data dependence analysis since it allows SAPFOR to perform the pairwise comparison of subscript expressions which calculate the addresses of accessed elements [24]. Unfortunately, variable dimension sizes and loop bounds may prohibit such comparison because C language does not ensure that subscript expression is in bounds value of an array dimension. That is why we introduce an analysis option which allows the user to force data dependence analysis and to assume that subscript expression is in bounds value.

Although SAPFOR is not able to reveal privatizable arrays in a static way, it uses dynamic analysis [14] which is helpful in case of EP and BT programs.

The EP benchmark has two features that require a manual program transformation. Firstly, it uses a reduction array, the use of which in C-DVMH is currently not supported. This array consists of 10 elements and we manually replaced it with 10 scalar variables. In this case, SAPFOR was able to automatically detect the presence of reduction operations and it generated the corresponding DVMH directives. Secondly, this benchmark uses a privatizable array of a very large size; this prevents the execution of the program on GPU. In order to eliminate this array, we manually fused two adjacent loops into a single loop (the first loop initializes this array and the second loop accesses the calculated values) and added a re-calculation of the required elements (two neighboring array elements) at each iteration of the new loop. As a result, the array was replaced with two scalar variables.

Each of benchmarks uses time measurement functions which access global variables to store execution time. If the calls of these functions were placed inside the loop body, this loop could not be parallelized. Whether the control flow reaches these calls depends on the program input, so static analysis techniques are not able to parallelize this loop. We removed unreachable calls manually if SAPFOR detected the likelihood of a side effect which prevents parallelization.

4.2 Performance Results

Table 2 shows the execution times of parallel versions obtained after automatic parallelization of the transformed benchmarks. The benchmarks have been also parallelized using OpenMP and OpenCL manually [19]. The results of corresponding launches are also given.

It can be noted that the sequential transformed version of the BT benchmark is better optimized than the original one. It is the result of inline expansion which provides the compiler with more optimization opportunities. On the other hand, the elimination of the private array in the EP benchmark increases the amount of computations, and as a result, slows down the sequential program.

A significant advantage of the OpenCL version of the CG benchmark compared to the DVMH version is caused by the use of shared memory on GPU. In addition, in the OpenCL version, the developers performed the vectorization of some inner loops. At the same time, DVMH and OpenCL versions of EP benchmark have similar performance, and on the BT benchmark, the DVMH program significantly outperforms OpenCL version. As to the maintainability, the OpenCL program is dramatically inferior to DVMH program because it is very different from the original sequential program.

Table 2. The execution time(s) of the NAS Parallel Benchmarks (NPB)

Benchmark		Sequential		SAPFOR		Manual	
				DVMH		OpenMP	OpenCL
Name	Class	Original	Transformed	CPU	GPU	CPU	GPU
BT	A	39.71	38.75	8.21	9.65	8.3	21.29
	B	169.72	161.49	34.3	34.83	35.71	77.15
	C	720.86	696.74	145.72	127.16	149.85	356.70
CG	A	0.82	0.83	0.33	0.42	0.23	0.07
	B	75.99	75.63	15.2	10.74	14.63	2.0
	C	213.11	222.67	40.14	46.67	39.05	6.45
EP	A	15.83	18.56	1.77	0.53	1.64	0.4
	B	63.2	74.27	7.07	1.49	6.55	1.42
	C	252.94	297.07	28.28	5.35	26.04	5.05

5 Related Work

There is a large number of studies dedicated to the automation of parallel programming. In this section we will consider some of them.

Polly [6] focuses on loop transformations to optimize data-locality and to exploit OpenMP level parallelism as well as to vectorize loops. It relies on a polyhedral model to optimize the program. Polly-ACC [7] extends Polly to bring accelerator support to generated parallel programs. Although the polyhedral model has a high potential for detecting parallelism in the program, it imposes significant restrictions on the source code that can be processed. The low level of LLVM IR, which is used to analyze and transform programs, does not allow the programmer to update or even view the generated code. Moreover, polyhedral based transformations implemented in a source-to-source way as in Plutto [25] produces the code which is significantly different from the original one and it can be quite difficult for the user to maintain it.

The application of the static analysis in these tools limits the possibilities of parallelization. In some cases this analysis does not allow one to estimate the sizes of the array dimensions and of the loop boundaries. Thus, the absence of such information may lead to a conservative assumption of the presence of data dependencies. Unlike Polly and the Polly-ACC, the Apollo [26] optimizer, which also relies on the polyhedral model, applies speculative optimizations at run time. However, Apollo does not allow parallelization of programs for GPU.

DiscoPop [8] relies on dynamic profiling information to reveal task graph which can be transformed with both loop-level and task-level parallelism. Clang is used to perform source-to-source transformation and to obtain parallel code using Intel Threading Building Blocks (TBB). It does not imply a significant transformation of the original program to increase the available parallelism. For each task in the task graph the corresponding code section is taken from the source code. Then it is wrapped up in a separate function which becomes a node in a flow graph of a parallel program.

The authors present the results of the automatic parallelization of programs from NAS Parallel Benchmarks (loop-level parallelism was exploited). Despite the use of dynamic analysis techniques, the performance of the parallel code is quite low and significantly inferior to the manually parallelized programs. Investigation of task-level parallelism is mainly suitable for programs with an unchangeable flow graph, such as application of different filters in image processing. In the case of repeated rebuilding of the flow graph, the overhead is very high.

6 Conclusion

The paper proposes an approach to the automation of parallel programming which follows an implicit parallel programming methodology. This approach was implemented in SAPFOR which includes an automatic parallelizing compiler. SAPFOR also provides source-to-source transformation techniques that allow the user to bring the sequential program to a well-formed version. SAPFOR relies on DVMH directive based programming model to exploit loop-level parallelism for multi-core processors and accelerators. Based on the proposed approach we perform semi-automatic parallelization of some applications from the C version of the NAS Parallel Benchmarks. The paper shows that automatically generated parallel versions have the similar performance to the manually parallelized ones.

This paper advocates the use of low-level program transformations, which are invisible to the programmer, to increase the quality of the analysis of the original program. We propose a novel approach that enables property sensitive transformations. This means that the internal representation of the program can be transformed to the most suitable form for program analysis.

The evaluation results show that SAPFOR is still suffering from an insufficient level of inter-procedural analysis. Future work will focus on this issue. We also intend to increase the number of supported C-DVMH constructions to distribute the data and the computations between several accelerators and nodes of the heterogeneous cluster.

References

1. Konovalov, N.A., Krukov, V.A., Mikhajlov, S.N., Pogrebtsov, A.A.: Fortan DVM: a language for portable parallel program development. Program. Comput. Softw. **21**(1), 35–38 (1995)
2. Bakhtin, V.A., Klinov, M.S., Krukov, V.A., Podderugina, N.V., Pritula, M.N., Sazanov, Yu.L.: Extension of the DVM-model of parallel programming for clusters with heterogeneous nodes. Bull. South Ural State Univ. Ser. Math. Model. Program. Comput. Softw. **18(277)**(12), 82–92 (2012). (in Russian)
3. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.P.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, pp. 519–530 (2013)
4. Beaugnon, U., et al.: VOBLA: a vehicle for optimized basic linear algebra. In: Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2014, New York, NY, USA, pp. 115–124 (2014)
5. Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., Amarasinghe, S.: Graphit: a high-performance graph DSL. In: Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 121:1–121:30 (2018)
6. Grosser, T., Groesslinger, A., Lengauer, C.: Polly-performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. **22**(04), 1250010 (2012)
7. Grosser, T., Hoefler, T.: Polly-ACC transparent compilation to heterogeneous hardware. In: ICS 2016: Proceedings of the 2016 International Conference on Supercomputing, June 2016, pp. 1–13 (2016). https://doi.org/10.1145/2925426.2926286
8. Zhao, B., Li, Z., Jannesari, A., Wolf, F., Wu, W.: Dependence-based code transformation for coarse-grained parallelism. In: Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores, San Francisco, CA, USA, pp. 1:1–1:10. ACM, February 2015

9. Kim, M., Kim, H., Luk, C.-K.: Prospector: a dynamic data-dependence profiler to help parallel programming. In: 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar 2010) (2010)
10. Garcia, S., Jeon, D., Louie, C., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. ACM SIGPLAN Not. (2011). https://doi.org/10.1145/1993316.1993553
11. Hwu, W.-M., et al.: Implicitly parallel programming models for thousand-core microprocessors. In: Proceedings of the 44th Annual Design Automation Conference (DAC 2007), pp. 754–759. ACM, New York (2007). https://doi.org/10.1145/1278480.1278669
12. Vandierendonck, H., Rul, S., De Bosschere, K.: The Paralax infrastructure: automatic parallelization with a helping hand. In: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE (2010)
13. Klinov, M.S., Krukov, V.A.: Automatic parallelization of Fortran programs. Mapping to cluster. In: Vestnik of Lobachevsky University of Nizhni Novgorod, no. 2, pp. 128–134. Nizhni Novgorod State University Press (2009). (in Russian)
14. Kataev, N., Smirnov, A., Zhukov, A.: Dynamic data-dependence analysis in SAPFOR. In: CEUR Workshop Proceedings, vol. 2543, pp. 199–208 (2020)
15. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California (2004)
16. Kataev, N.: Application of the LLVM compiler infrastructure to the program analysis in SAPFOR. In: Voevodin, V., Sobolev, S. (eds.) RuSCDays 2018. CCIS, vol. 965, pp. 487–499. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05807-4_41
17. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Boston (2006). p. 1038, Chap. 9
18. OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accessed 14 Apr 2020
19. 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)
20. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html. Accessed 14 Apr 2020
21. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Trans. Software Eng. **28**(12), 1146–1170 (2002). https://doi.org/10.1109/TSE.2002.1158288
22. Havlak, P., Kennedy, K.: An implementation of interprocedural bounded regular section analysis. IEEE Trans. Parallel Distrib. Syst. **2**(3), 350–360 (1991). https://doi.org/10.1109/71.86110
23. Goff, G., Kennedy, K., Tseng, C.-W.: Practical dependence testing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI 1991), pp. 15–29. ACM, New York (1991). https://doi.org/10.1145/113446.113448
24. Kataev, N., Vasilkin, V.: Reconstruction of multi-dimensional arrays in SAPFOR. In: CEUR Workshop Proceedings, vol. 2543, pp. 209–218 (2020)

25. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (2008)
26. Caamano, J.M.M., Sukumaran-Rajam, A., Baloian, A., Selva, M., Clauss, P.: APOLLO: automatic speculative polyhedral loop optimizer. In: 7th International Workshop on Polyhedral Compilation Techniques (IMPACT), Stockholm, Sweden, January 2017 (2017)