Russian Supercomputing Days 2018

Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR

September 24, 2018 | Moscow

Nikita Kataev

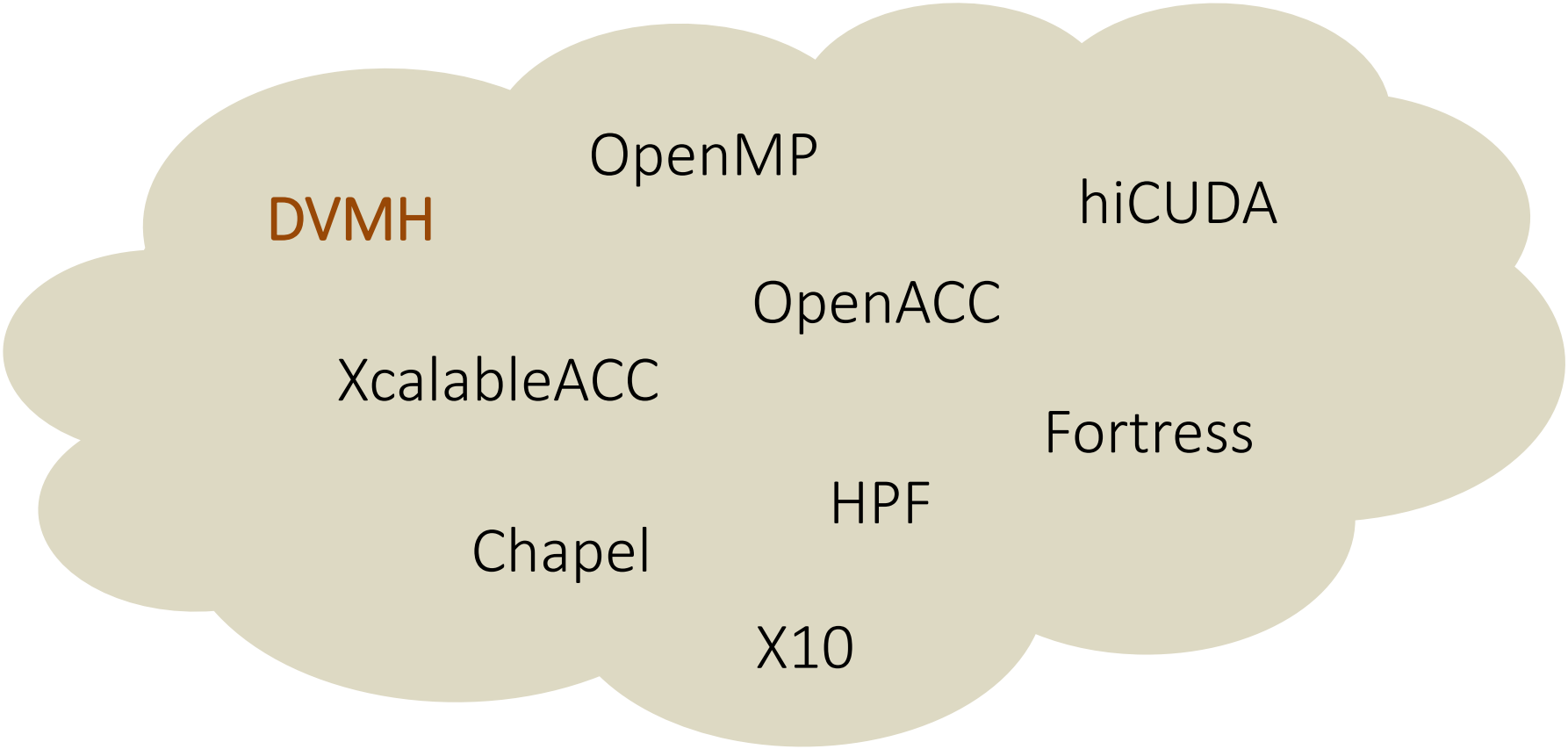
Keldysh Institute of Applied Mathematics RAS

Complexity of parallel programming

To effectively apply low-level programming models a programmer should fully understand hardware architecture as well as these models.

Simplification of parallel programming

Development of high-level parallel programming languages.



Simplification of parallel programming

Development of high-level parallel programming languages.

Directive-based programming models.

CUDA

OpenMP

MPI

DVMH

Directive-based programming model which aims to create parallel programs for heterogeneous computational clusters (GPU NVidia, Intel Xeon Phi, multicore CPUs).

The model includes two programming languages which are the extensions of standard C and Fortran languages by parallelism specifications: *CDVMH* and *Fortran-DVMH*

The parallel program is developed in terms of a sequential one.

Simplification of parallel programming

Development of high-level parallel programming languages.

CUDA

OpenM

MPI

```
#pragma dvm array distribute [block][block]
float A[L][M];
#pragma dvm array align([i][j] with A[i][j])
float B[L][M];

...

#pragma dvm region
{
#pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
for (i = 1; i < L - 1; i++)
  for (j = 1; j < M - 1; j++)
  {
    float tmp = fabs(B[i][j] - A[i][j]);
    eps = Max(tmp, eps);
    A[i][j] = B[i][j];
  }

#pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
for (i = 1; i < L - 1; i++)
  for (j = 1; j < M - 1; j++)
      B[i][j] = (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]) / 4.0f;
}
```

Simplification of parallel programming

SAPFOR (System For Automate Parallelization) is a software development suit that is focused on cost reduction of manual program parallelization.

The main goals of SAPFOR development:

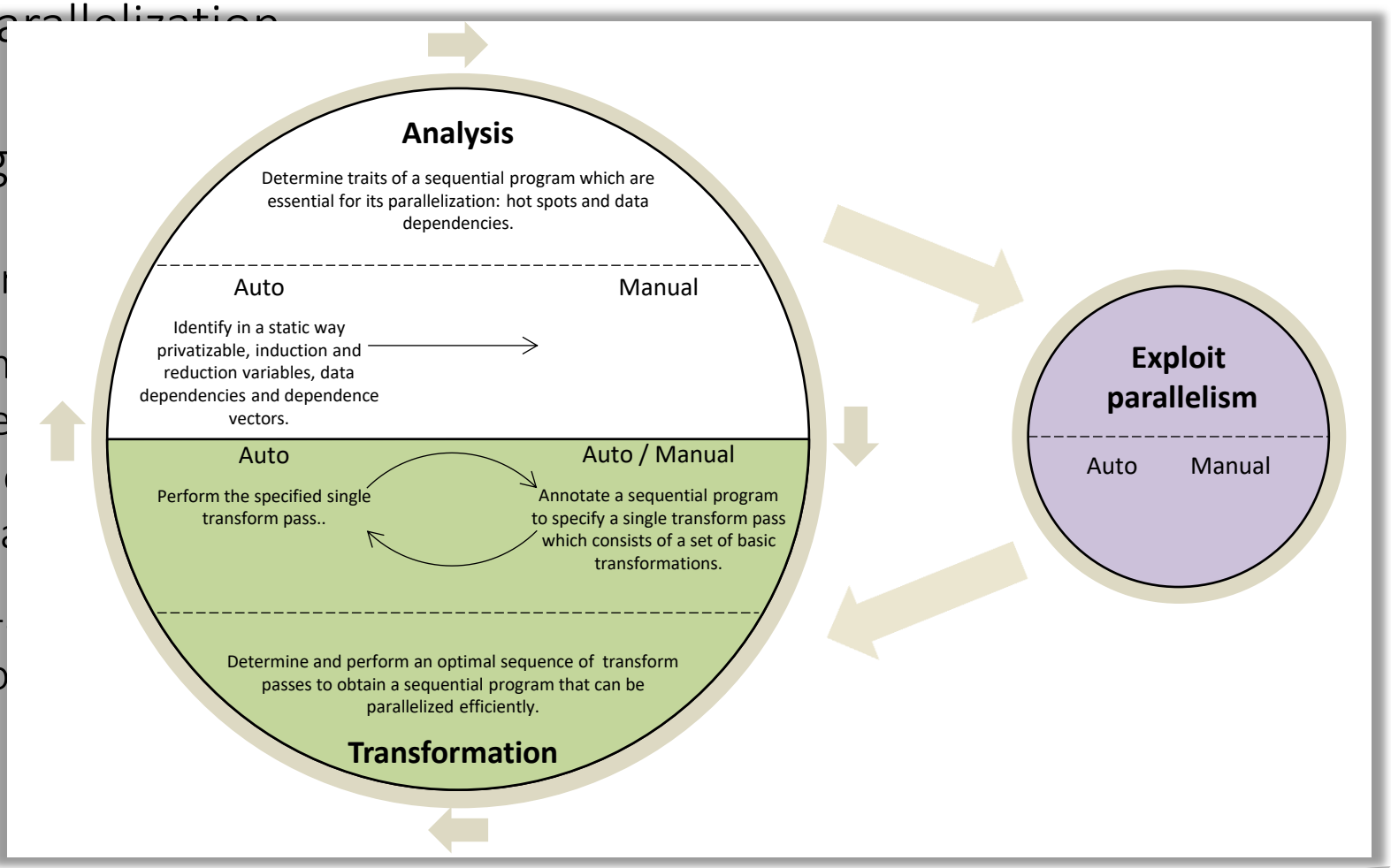
- Exploration of sequential programs (program analysis and profiling).
- Automatic parallelization (according to DVMH model) of a well-formed sequential program for which a programmer maximizes algorithm-level parallelism and asserts high-level properties (implicitly parallel programming).
- Semi-automatic program transformation to obtain a well-formed sequential version of the original program.

Simplification of parallel programming

SAPFOR (System For Automate Parallelization) is a software development suit that is focused on cost reduction of manual program parallelization

The main g

- Explor
- Auton
  seque
  paralle
  progra
- Semi-
  versio



SAPFOR Intermediate Representation (IR)

Requirements for SAPFOR IR implementation:

- Support for Fortran (95 and higher) and C (99 and higher) programming languages.
- Ability to implement different analysis technics (data dependence analysis, induction variable recognition and substitution, reduction variable recognition, privatization, points-to analysis, alias analysis) which are necessary for program parallelization.
- Support for source-to-source program transformation.
- Ability to represent large-scale computational applications. For example, front-end must support standard compiler options to simplify the use of build automation tools.

The following compiler infrastructures have been considered:
Sage, Cetus, Rose, Ops, GCC, LLVM.

The LLVM Compiler Infrastructure

The **LLVM** Project is a collection of modular and reusable compiler and toolchain technologies.

LLVM began as a research project and now it is an open source project. LLVM is widely used in academic, open source and commercial projects (Apple, Intel, NVIDIA, PGI and other). LLVM consists of a number of subprojects which aims to program optimization, program parallelization and program analysis.

LLVM operates on its own low-level code representation known as the LLVM intermediate representation (LLVM IR). Different front-ends can be used to compile multiple programming languages (C, C++, Fortran, Ada, Go and other).

LLVM provides a friendly API for designing analysis and transform passes. LLVM is currently written using C++ 11 conforming code. The LLVM libraries are well documented.

http://llvm.org/

Features of SAPFOR IR

Clang + LLVM (C/C++) and Sage/Flang + LLVM (Fortran)

Program parallelization in SAPFOR is a search for an optimal program-specific sequence of optimization passes:

- analysis passes compute information about the program.
- transform passes can use analysis passes to “improve” the original program.

LLVM optimization features are also implemented as a passes. It is possible to use existent passes as well as to implement the new ones.

Source-tor-source program transformation is based on Clang and Sage/Flang capabilities.

Meta information is used to utilize analysis results to evaluate a program in a higher level language. Correspondence between low-level and high-level program representations is maintained by synchronization points between the most important structures:

- loop tree;
- memory representation;
- memory access.

There is no need to achieve the complete correspondence between AST and LLVM IR.

Three levels of IR

```
int f(int x, int y) {
  return x + y;
}
```

```
`-FunctionDecl f 'int (int, int)'
  |-ParmVarDecl x 'int'
  |-ParmVarDecl y 'int'
  `-CompoundStmt
    `-ReturnStmt
      `-BinaryOperator 'int' '+'
        |-ImplicitCastExpr 'int' <LValueToRValue>
        | `-DeclRefExpr 'int' lvalue ParmVar 'x' 'int'
        `-ImplicitCastExpr 'int' <LValueToRValue>
          `-DeclRefExpr 'int' lvalue ParmVar 'y' 'int'
```

AST

LLVM IR

```
define i32 @f(i32 %x, i32 %y) {
entry:
  call void @llvm.dbg.value(metadata i32 %y, i64 0, metadata !11, metadata !12)
  call void @llvm.dbg.value(metadata i32 %x, i64 0, metadata !14, metadata !12)
  %add = add nsw i32 %x, %y
  ret i32 %call
}
```

```
...
!7 = distinct !DISubprogram(name: "f", scope: !1, file: !1, l
...
!10 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_
!11 = !DILocalVariable(name: "y", arg: 2, scope: !7, file: !1, line: 3, type: !10)
...
!13 = !DILocation(line: 3, column: 18, scope: !7)
!14 = !DILocalVariable(name: "x", arg: 1, scope: !7, file: !1, line: 3, type: !10)
```

DWARF

Three levels of IR

```
int f(int x, int y) {
  return x + y;
}
```

```
`-FunctionDecl f 'int (int, int)'
  |-ParmVarDecl x 'int'
  |-ParmVarDecl y 'int'
  `-CompoundStmt
    `-ReturnStmt
      `-BinaryOperator 'int' '+'
        |-ImplicitCastExpr 'int' <LValueToRValue>
        | `-DeclRefExpr 'int' lvalue ParmVar 'x' 'int'
        `-ImplicitCastExpr 'int' <LValueToRValue>
          `-DeclRefExpr 'int' lvalue ParmVar 'y' 'int'
```

AST

LLVM IR

```
define i32 @f(i32 %x, i32 %y) {
entry:
  call void @llvm.dbg.value(metadata i32 %y, i64 0, metadata !11, metadata !12)
  call void @llvm.dbg.value(metadata i32 %x, i64 0, metadata !14, metadata !12)
  %add = add nsw i32 %x, %y
  ret i32 %call
}
```

```
...
!7 = distinct !DISubprogram(name: "f", scope: !1, file: !1, l
...
!10 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_
!11 = !DILocalVariable(name: "y", arg: 2, scope: !7, file: !1, line: 3, type: !10)
...
!13 = !DILocation(line: 3, column: 18, scope: !7)
!14 = !DILocalVariable(name: "x", arg: 1, scope: !7, file: !1, line: 3, type: !10)
```

DWARF

Goals of program transformation

Program transformation which improves the quality of the source program analysis:

- IR-level transformation;
- original program keeps its properties across the transformation;
- hidden from a programmer.

Before	After

1. void copy(float *A, int N, int M) {
2. int X;
3. for (int I = 0; I < N; ++I) {
4. X = I + M;
5. A[I] = A[X];
6. }
7. }
	1. void copy(float *A, int N, int M) {
	2. for (int I = 0; I < N; ++I) {
	3. A[I] = A[I + M];
	4. }
	5. }

Program transformation which reveals hidden parallelism in a the source code:

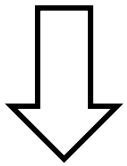
- source-to-source transformation;
- changes properties of the original program;
- a programmer may evaluate SAPFOR decisions and make corrections.

Memory access description

This includes results of data dependence analysis, induction and reduction variable recognition, privatization, alias analysis.

All information provided by SAPFOR must be attached to the items of the source program.

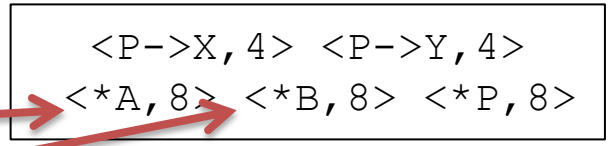
Lower level of LLVM IR does not directly allow LLVM to be applicable for the description of analysis results.



Source-level alias tree has been develped.
It depicts the structure of accessed memory using source-level debug information.

Source-level alias tree: example

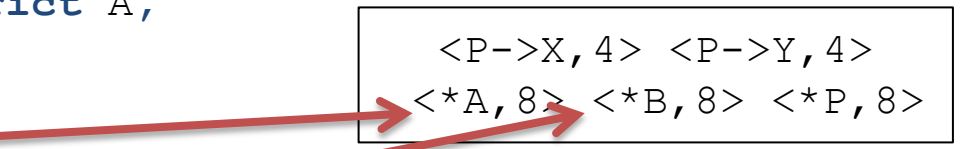
```
struct S {float X; float Y;};
void foo(struct S * restrict A,
struct S * restrict B) {
  struct S *P;
  P = A;
  P->X = 0;
  P->Y = 0;
  P = B;
  P->X = 0;
  P->Y = 0;
}
```



- Each memory location is identified by the address of the start of the location and its size.
- Two memory locations fall into a single node of a source-level alias tree, if they may alias.
- The union of all the memory locations from the parent nodes covers the union of the memory locations from a child

Source-level alias tree: example

```
struct S {float X; float Y;};
void foo(struct S * restrict A,
struct S * restrict B) {
  struct S *P;
  P = A;
  P->X = 0;
  P->Y = 0;
  P = B;
  P->X = 0;
  P->Y = 0;
}
```

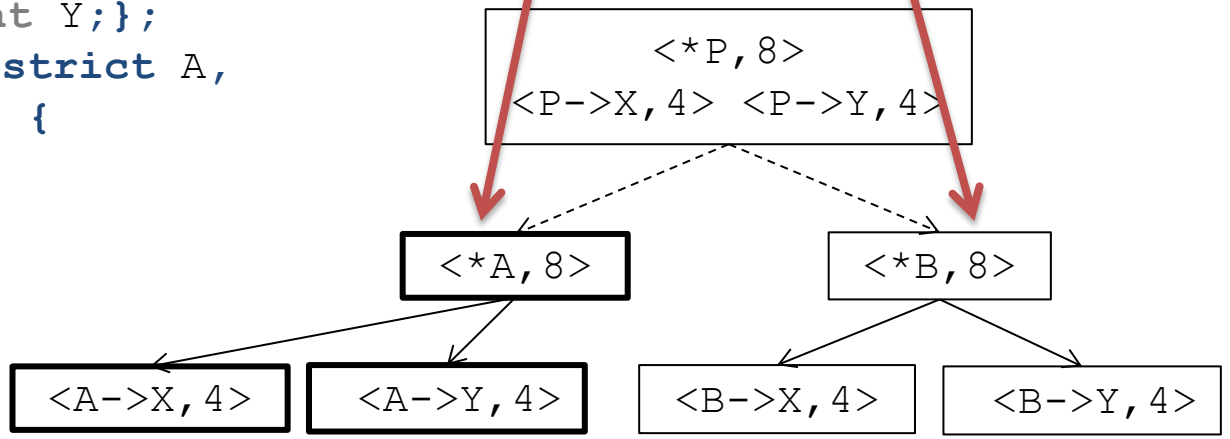


IR-level transformation

```
struct S {float X; float Y;};
void foo(struct S * restrict A,
struct S * restrict B) {

  A->X = 0;
  A->Y = 0;

  B->X = 0;
  B->Y = 0;
}
```



Source-level alias tree

The source-level alias tree restores original program properties after transformation. It depicts the structure of accessed memory using source-level debug information:

- summarizes IR-level memory locations to higher level items;
- corresponds to a hierarchical type system of a higher level language (aggregate type holds its member, ‘long’ holds ‘int’, etc.).
- does not directly depend on a programming language and front-end, because it uses metadata (DWARF), rather than abstract syntax tree;
- adjust its structure across the transformation of LLVM IR which does not affect the structure of the memory used in the original program.

Evaluation of the NAS Parallel Benchmarks (NPB)

Benchmark	Number Of Loops										
	Total	Array	Call	Indep.	Dep.			Priv.		Ind.	Red.
					Total	Call	Only	Total.	Indep.		
BT	181	171	51	101	80	50	16	109	65	179	0
CG	47	14	6	17	30	6	2	15	1	46	8
EP	9	6	5	3	6	5	2	3	0	9	2
IS	12	11	3	4	8	3	2	3	0	12	0
LU	187	156	40	96	91	39	27	84	39	171	3
MG	81	37	22	12	69	19	14	38	2	77	1
SP	250	243	48	158	92	47	16	145	87	248	0
Total	767	638	175	391	376	169	79	407	194	742	14

- Loops with data dependencies averaged 49% (Dep.) of the total number of loops.
- A large number of dependencies indicate the need to program transformation for their parallel execution.
- The variable privatization is necessary for the parallel execution of about the half of the loops (including loops without dependencies).

Conclusion

SARFOR uses:

- LLVM IR to obtain information about the original program;
- AST for source-to-source program transformation.

Investigation of transformed LLVM IR improves the quality of the source program analysis.

The developed source-level alias tree allows us to restore original program properties after transformation.

Source-level transformations are vital to provide program parallelization in interaction with a programmer. So, the future works involve the implementation of source-to-source transformations and application of LLVM based analysis to check their correctness.

Thank you for attention

Russian Supercomputing Days 2018

http://dvm-system.org