CSIT (June 2019) 7(2):105–110
https://doi.org/10.1007/s40012-019-00248-5

S.I. : VISVESVARAYA

An analysis of executable size reduction by LLVM passes

Shalini Jain[1] · Utpal Bora[1] · Prateek Kumar[1] · Vaibhav B. Sinha[1] · Suresh Purini[2] · Ramakrishna Upadrasta[1]

Received: 3 December 2018 / Accepted: 27 May 2019 / Published online: 3 June 2019

Abstract The formidable increase in the number of smaller and smarter embedded devices has compelled programmers to develop more and more specialized application programs for these systems. These resource intensive programs that have to be executed on limited memory systems make a strong case for compiler optimizations that reduce the executable size of programs. Standard compilers (like LLVM) offer an out-of-the-box `-Oz` optimization option—just a series of compiler optimization passes—that is specifically targeted for the reduction of the generated executable size. In this paper, we aim to analyze the effects of optimizations of LLVM compiler on the reduction of executable size. Specifically, we take the size of the executable as a metric and attempt to divide the `-Oz` series into logical groups and study their individual effects; while also study the effect of their combinations. Our preliminary study over SPEC CPU 2017 benchmarks gives us an insight into the comparative effect of the groups of passes on the executable size. Our work has potential to enable the user to tailor a custom series of passes so as to obtain the desired executable size.

Keywords Compilers · Compiler optimizations · Code size optimizations

An earlier version of this work was peer reviewed and selected for presentation at EuroLLVM 2018.

✉ Ramakrishna Upadrasta
ramakrishna@iith.ac.in

Shalini Jain
cs15resch11010@iith.ac.in

Utpal Bora
cs14mtech11017@iith.ac.in

Prateek Kumar
cs15btech11031@iith.ac.in

Vaibhav B. Sinha
cs15btech11034@iith.ac.in

Suresh Purini
suresh.purini@iiit.ac.in

[1] Computer Science and Engineering, Indian Institute of Technology, Hyderabad, India

[2] Computer Science and Engineering, International Institute of Information Technology, Hyderabad, India

1 Introduction

The ever increasing demand for smart embedded devices has lead to the development of more sophisticated applications for these memory constrained devices. One of the major roadblocks in the execution of such programs on embedded devices is dealing with the meagre amount of available memory. While this has forced programmers to develop programs with smaller executables, it has also forced compiler writers to look for optimizations to reduce the executable size. The `Oz` optimization level of the LLVM compiler [1] is created for this specific need, similar to the `O1`, `O2`, `O3` levels that target reduction of running time.

In LLVM 5.0, the `Oz` optimization has a total of 218 passes, of which 70 are transformation passes, some of which are scheduled based on requirements by other passes. Currently, it is clearly not known which passes, or groups of passes, have the highest effect on the executable size. In our work, we disassemble the series of passes of `O1` and observe the results of the groups of these passes on executable size.

We first make a comparative study of the executable size obtained by the standard optimization

levels—O0, O1, O2, O3, Os, Oz—offered by LLVM on standard benchmarks (SPEC CPU 2017 [2]). Then, we break the series of transformation passes in Oz level into various logical groups; each group broadly performs a specific variety of optimization, like as canonicalization, loop optimization, dead code elimination, vectorization, etc. We follow the above to study of the effect of these groups as well as combinations of groups on executable size.

In our study, we attempt to give new insights into the following two important questions related to executable-size:

- Which group of passes have the most pronounced effect on the executable size?
- Is it possible to break down the specific groups of passes in LLVM to spot the exact (subset of) passes, that have the highest effect on code bloat?

We conclude our study by formulating empirical results of the trade-off between reduction (in executable size) versus run-time obtained by groups and their combinations.

Our results can be used to tailor a cosmetic series of passes which can be used for highly memory critical applications. The results of our work on the effect of groups of passes thus provides an end-to-end answer for programmers seeking for compiler based executable size reduction. We also believe that our work has potential to allow one to choose the set of passes that substantially improve the executable size, while losing some amount of running-time.

The following is the organization of our paper: In Sect. 2, we show comparison of the basic optimization flags in term of executable binary size and execution time. In Sect. 3, we propose a grouping of the passes. In Sect. 4, we show the results for binary code-bloat and run-time. In Sect. 5, we perform a sub-grouping on group G0 for checking the performance of individual optimizations. In Sect. 6, we conclude our work along with discussing some limitations and future work.

2 Comparison of the basic optimization flags

In LLVM, O0, O1, O2, O3, Os and Oz are standard optimization flags, extensively used for the optimization of C and C++ programs. These optimizations mainly target the objectives of improving the run-time of the compiled binary and also reducing the size of the binary.

In this section, we perform a comparison of these optimization flags on SPEC CPU 2017 Benchmarks. For this, we compile the SPEC CPU 2017 benchmark suite with the flags O0, O1, O2, O3, Os and Oz and obtain the total binary size, and its run time. For this experiment we use an

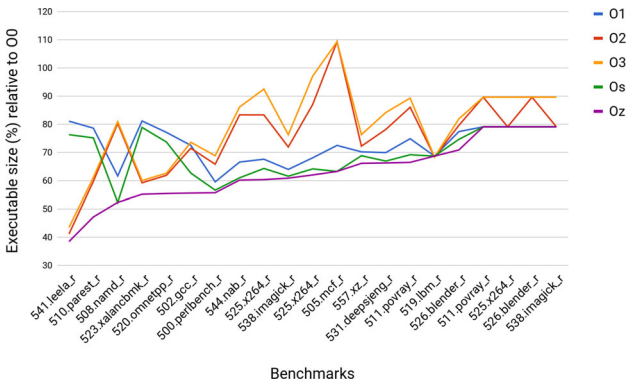

Fig. 1 Executable-size comparison versus O0 (SPEC CPU 2017)

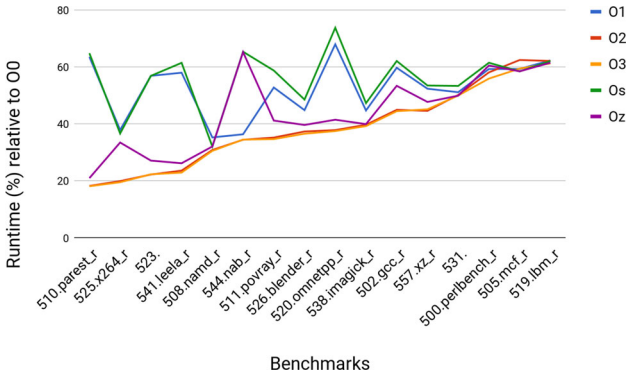

Fig. 2 Run-time comparison versus O0 (SPEC CPU 2017)

Intel 2.30 GHz *x*86_64 machine. Figures 1 and 2 represents the sizes of the binary and run time of the benchmark codes for the different optimization flags respectively.

It can be observed from Fig. 2 that the run time of O3 is lowest among all. This is justified as O3 concentrates on only optimization with run time as metric. It can also be seen that the run time produced by Oz is often not very high (except a few cases) as compared to O3.

In Fig. 1, it can be seen that, in general, executable size is highest for O3 followed by O2. Oz generally has the smallest executable size followed by Os. This can be explained as O3 concentrates mainly on run-time optimizations and gives lesser importance to executable size. But it should be noted here that executable size is not completely neglected by O3; we will see later in Sect. 4, that executable size still reduces when compared to no optimization (option O0).

Average executable size and average run-time relative to O0 for different optimization levels are shown in Figs. 3 and 4 respectively. Figure 5 represents the pareto graph for run-time and executable size relative to O0 for SPEC CPU 2017 benchmarks.

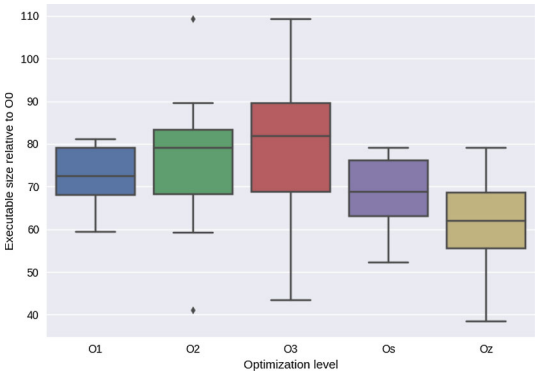

Fig. 3 Executable-size versus optimization levels (SPEC CPU 2017)

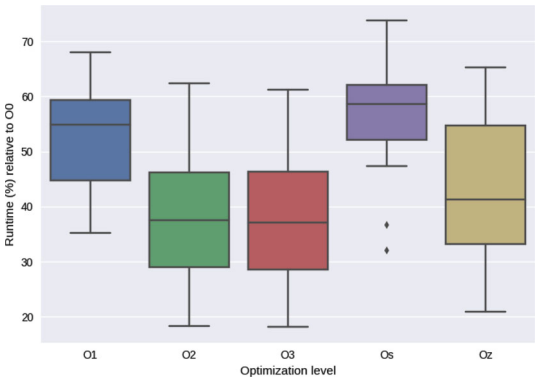

Fig. 4 Run time versus optimization levels (SPEC CPU 2017)

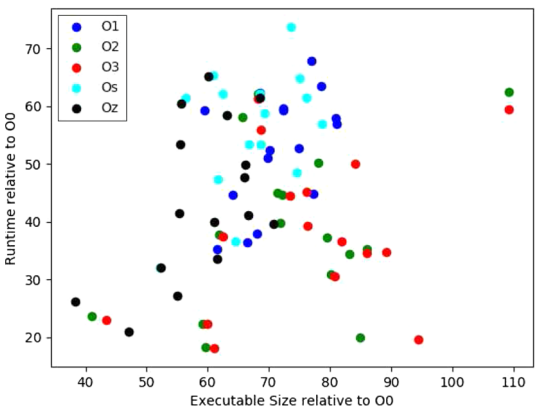

Fig. 5 Run time versus executable-size (relative to -O0)

3 Grouping of passes

Before we begin to obtain the values of run-time and executable-size for each pass, we need to group the passes. This is primarily because it is hard to run all the passes of `Oz` individually, as they are huge in number. Also, many passes are not standalone passes; they are enabled only after some other optimizations. Moreover, some of them are mandatory prerequisite passes and hence must always be run.

So, for the reasons mentioned above, we attempt to group the passes of `Oz` logically so that each group broadly performs some task. The passes considered here are only transformation passes and not analysis passes; the transformation pass automatically calls its corresponding analysis passes in a demand driven manner.

3.1 Formation of groups

For breaking the series of passes into groups, we choose the series of passes of `Oz`. Table 1 contains all the groups that we constructed from `Oz` optimization sequence.

Changing the order of the passes may disrupt the series of optimization intended, so we merely break the series at logical positions. Now we briefly describe each group:

The group `GF` forms the function pass set, which is applied at function scope. While the rest from `G0`–`G5` form the module pass series.

`GF`: This is a set of very important and the most influential passes which are applied at function scope. These are default set of passes which are run in all our experiments whenever we run any of `G0`–`G5`.

`G0`: It contains the most elementary passes. This is a compulsory group, which means we do not run any group without running `G0` first. The major passes in this group include memory to register pass, dead argument elimination pass, etc. This group also contains the compulsory pass inline (which is the only proper transformation pass which appears in `O0`, i.e. no optimization).

`G1`: This is a general group of passes, the major ones among them involved in SSA form conversion. Moreover, it has other exciting passes like tail call elimination, etc.

`G2`: This group concentrates on loop optimizations like loop rotate, loop invariant code motion, loop unswitch, loop deletion, loop unrolling, etc. Moreover, it has other passes that aid in this process like reassociate, induction variable pass, etc.

`G3`: This group is broadly involved in dead code elimination and so contains passes like dead store elimination, aggressive dead code elimination, bit-tracking dead code elimination, etc.

`G4`: This group concentrates on loop optimizations and vectorization. It contains passes like loop rotate, loop distribute, loop vectorize, slp vectorize, loop unroll, loop invariant code motion etc.

`G5`: This is the last group broadly involved in dead code elimination. It consists of Strip dead prototypes, global dead code elimination, constant merging, etc.

Table 1 Different groups of passes

Group name	Sequence of passes
GF	-simplifycfg -sroa -early-cse -lower-expect
G0	-forceattrs -inferattrs -ipsccp -globalopt -mem2reg -deadargelim -instcombine -simplifycfg -prune-eh -inline -functionattrs
G1	-sroa -early-cse-memssa -speculative-execution -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg
G2	-reassociate -loop-rotate -licm -loop-unswitch -simplifycfg -instcombine -indvars -loop-idiom -loop-deletion -loop-unroll
G3	-mldst-motion -gvn -memcpyopt -sccp -bdce -instcombine -jump-threading -correlated-propagation -dse -adce -licm -simplifycfg -instcombine -elim-avail-extern -rpo-functionattrs -float2int
G4	-loop-rotate -loop-distribute -loop-vectorize -loop-load-elim -instcombine -latesimplifycfg -instcombine -loop-unroll -instcombine -licm -alignment-from-assumptions
G5	-strip-dead-prototypes -globaldce -constmerge -loop-sink -instsimplify -simplifycfg

All optimization levels consist of the verify pass at the end of optimization which verifies the sanity of the optimized IR. We do not add it to any group since it is a compulsory pass and must be run at the end of each series of groups we consider, either individual or combination.

3.2 Combination of groups

We experimented the benchmarks on individual groups as well as the combination of groups. While forming the combinations, we take care of the following points:

- G0 is a compulsory (prerequisite) group, and we always run it first even when we run individual groups. Since it is compulsory we do not mention it when we name the groups; it is considered implicitly present.
- While forming groups, we respect the order of the groups, i.e. we form combinations such as `G3G4` but not like `G4G3` because this is against the natural order of the passes.

4 Results

We have done the experiments on the SPEC CPU 2017 benchmark on an Intel Machine. This section shows the results obtained from individual groups and their combinations.

4.1 Results of binary code bloat

First, we plot the relative sizes of binary as compared to no optimizations (`O0`). We plot relative sizes since the binary sizes are quite large and the graphs are not very clear if we plot directly.

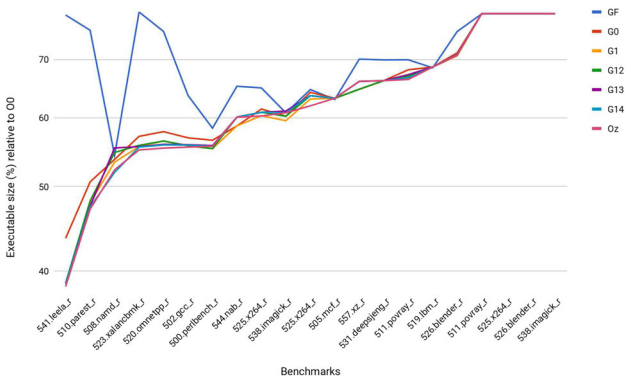

Fig. 6 Executable size: serial application of groups (SPEC CPU 2017)

From Fig. 6 we observe that `Oz` reduces the binary code size as compared to `O0`. The major reduction of code size comes by the application of `GF` and `G0`. `G1` further reduces the code size, but all the other groups increase the binary size. This claim becomes clear from Fig. 7. In Fig. 7, we plot the relative code bloat of all groups against `G0`. We see that apart from `G1` no other group contribute much. Note that here each group first runs on `G0` and then itself. So we conclude that `G0` passes do the majority of shrinkage while the other groups do not have much effect. Note that `G0` contains the compulsory passes like `mem2reg`, etc, which are applied in all levels of optimizations. So we can safely predict that its effect will be reflected in all the optimization levels. Moreover, we also note that `G1` was majorly concentrating on passes like SSA form conversion, etc. Thus we may say that SSA and these passes together reduce the code size.

From Figs. 6 and 8, we make the following conclusion: as the number of passes increases the individual impact of

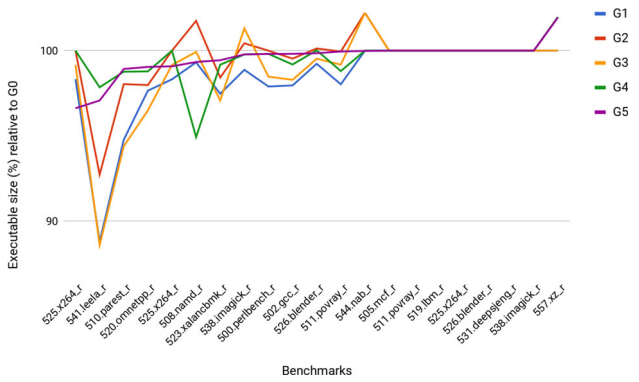

Fig. 7 Effect of individual groups

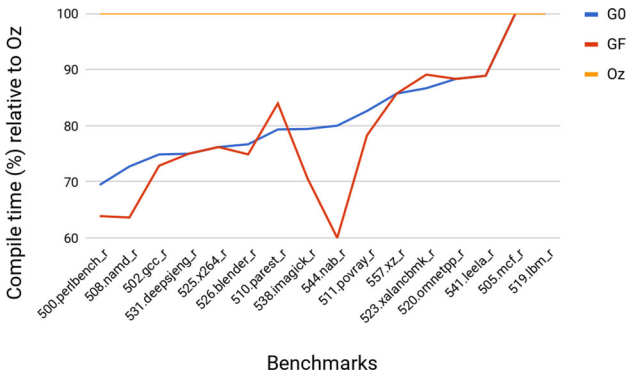

Fig. 9 Compile time: effective groups (SPEC CPU 2017)

passes gets blurred. The combination of groups slowly converge to the size of `Oz` as the size of the groups increases. This was evident from Fig. 6 the code bloat obtained with only `G1`, `G1G2`, `G1G2G3` etc.

4.2 Results of run time

Again, as mentioned previously we obtain the run time using optimizations from individual groups and their combinations. This subsection presents these results. First, we plotted the relative run time of individual groups as compared to no optimizations (`O0`). We plot relative percentage since the run time varies across different benchmarks.

From the Fig. 8, we observe that `Oz` takes the least run time which follows from the fact that it runs the highest number of passes. We see that among the individual groups `GF` gives the maximum reduction. This leads us to suspect that `GF` is very important for the overall run time. `G0` also helps for run time. `G1`, `G2`, `G3`, `G4` and `G5` almost have no effect on run time.

This confirms our claim that `GF` and `G0` contribute to the major section of optimization run time. We observe that as the size of the combination of groups increase, the run time decreases as a response to the increase in the number of passes. We also plotted the relative percentage of the compile time varies across different benchmarks in Fig. 9.

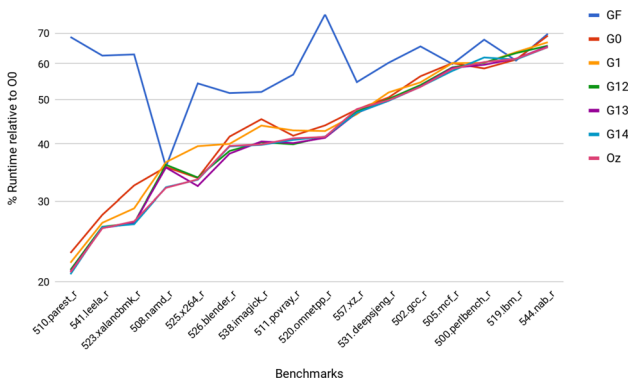

Fig. 8 Run time: serial application of groups (SPEC CPU 2017)

5 Sub-groups

From our above experiments, we saw that Group `G0` has a very significant impact on both the run time and executable size. So to pinpoint the exact few passes among the Group `G0` passes which are the most significant, we further break down Group `G0` into subgroups and observe their effects on executable size. Figure 10 shows the subgrouping of group `G0`.

Observe here that the subgrouping is not done in a way similar to grouping. Groups were disjoint parts of the full optimization series, but subgroups are not so. The reason for this is that we have to run `-instcombine` and `-simplifycfg` after any of the passes as they expect these passes after them.

Passes of G0	Sub-Group 1	Sub-Group 2	Sub-Group 3	Sub-Group 4
-forceattrs	-forceattrs			
-inferattrs	-inferattrs			
-ipsccp		-ipsccp		
-globalopt		-globalopt		
-mem2reg			-mem2reg	
-deadargelim			-deadargelim	
-instcombine	-instcombine	-instcombine	-instcombine	
-simplifycfg	-simplifycfg	-simplifycfg	-simplifycfg	
-prune-eh				-prune-eh
-inline				-inline
-functionattrs				-functionattrs

Fig. 10 The sub-groups of G0

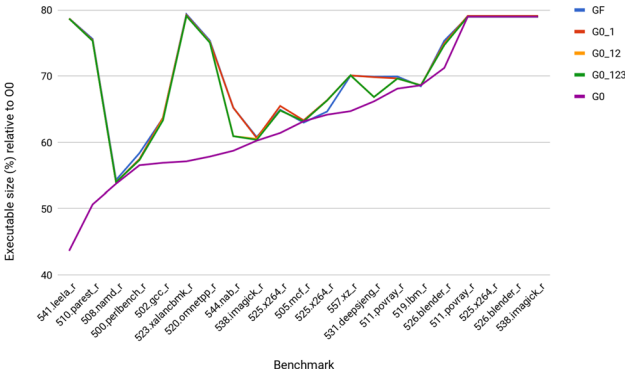

Fig. 11 Analysis of serial application of subgroups

We now present the results of executable size by these subgroups in Fig. 11. Note that before running the passes of any of the subgroups, the passes of `GF` are always executed.

Surprisingly we observe that a lot of optimization happens in `GF` itself. And the complete execution of `G0` is necessary to have a reasonable impact upon the executable size.

The observation of the impact of `GF` passes motivates us to break down `GF` further and examine the impact by the passes within it. Since there are only four passes in `GF`; we do not form subgroups, we just observe the impact of successive executions of the passes.

From Fig. 12, we clearly see the significance of the first pass of `GF`: `-simplifycfg`. It alone contributes to roughly 20% of reduction from O0. This is expected as this `-simplifycfg` would execute directly on the IR created by LLVM and so it would clean up a lot of code. Moreover the impact of the second pass: `-sroa` can also not be ignored. In fact, it alone contributes to about 10% additional decrease on executable size. The other two passes do not have much impact on executable size.

We however do not analyze the impact of the subgroups of `G0` and `GF` on run time. This is because these two groups have are essential and must always be run whenever we optimize code.

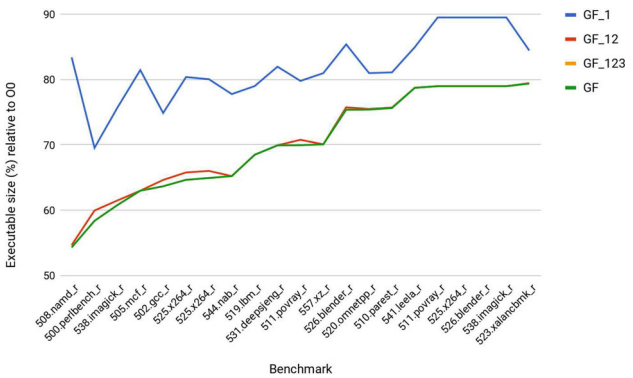

Fig. 12 Inside the function passes

Our experiments show the impact of the passes of `G0` and `GF` on executable size. We conclude from the experiments in Sects. 4 and 5 that we must always execute the `GF` and `G0` passes whenever we want to optimize.

6 Conclusions, limitations and future work

In this paper we examined the impact on run time and binary code size of the passes and optimization levels of LLVM on Intel Architecture. We proposed a logical grouping of passes and studied their impact on the run time and the binary size. We also subdivided the most significant groups to discover the exact passes having the highest impact.

The binary size for each combination of groups may not be present normally because of resource constraints. In such a case, it would be desirable to be able to predict the binary size obtained by combinations of groups, given the binary size of individual passes along with `O0` (unoptimized) and `O3`. In our future work, we would address the above problem by posing it as a regression problem.

Acknowledgements This work has been partially supported by the Project under *The Visvesvaraya Ph.D. Scheme of Ministry of Electronics & Information Technology*, Government of India, being implemented by Digital India Corporation (formerly Media Lab Asia). The work was also enabled by travels funded by *The LLVM Foundation*. The fellowship has helped in publishing research contributions in [3–6].

Funding Funding was provided by Department of Electronics and Information Technology, Ministry of Communications and Information Technology.

References

1. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04). IEEE Computer Society, Washington, p 75
2. https://www.spec.org/cpu2017/
3. Dangeti TK, Bora U, Das S, Grosser T, Upadrasta R (2017) Improved loop distribution in LLVM using polyhedral dependences. In: The fourth workshop on the LLVM compiler infrastructure in HPC. https://llvm-hpc4-workshop.github.io/
4. Jain S, Kumar K, Purini S, Das D, Upadrasta R (2017) An LLVM based loop profiler. US LLVM Developers' meeting. https://llvm.org/devmtg/2017-10/
5. Bora U, Grosser T, Upadrasta R (2016) PolyhedralInfo: polly as an analysis pass in LLVM. Google Summer of Code 2016 and Poster+Lightning Talk, US LLVM Developers' meeting
6. Bhatu Pratik (2015) Compile-time optimizations in polly. Google Summer of Code. https://www.googlemelange.com/archive/gsoc/2015/orgs/llvm/projects/bhatuzdaname.html