Fast Heuristic-Based GPU Compiler Sequence Specialization

Ricardo Nobre(✉), Luís Reis, and João M. P. Cardoso

Faculty of Engineering, University of Porto, INESC TEC, Porto, Portugal
{ricardo.nobre,luis.cubal}@fe.up.pt, jmpc@acm.org

Abstract. Iterative compilation focused on specialized phase orders (i.e., custom selections of compiler passes and orderings for each program or function) can significantly improve the performance of compiled code. However, phase ordering specialization typically needs to deal with large solution space. A previous approach, evaluated by targeting an x86 CPU, mitigates this issue by first using a training phase on reference codes to produce a small set of high-quality reusable phase orders. This approach then uses these phase orders to compile new codes, without any code analysis. In this paper, we evaluate the viability of using this approach to optimize the GPU execution performance of OpenCL kernels. In addition, we propose and evaluate the use of a heuristic to further reduce the number of evaluated phase orders, by comparing the speedups of the resulting binaries with those of the training phase for each phase order. This information is used to predict which untested phase order is most likely to produce good results (e.g., highest speedup). We performed our measurements using the PolyBench/GPU OpenCL benchmark suite on an NVIDIA Pascal GPU. Without heuristics, we can achieve a geomean execution speedup of 1.64×, using cross-validation, with 5 non-standard phase orders. With the heuristic, we can achieve the same speedup with only 3 non-standard phase orders. This is close to the geomean speedup achieved in our iterative compilation experiments exploring thousands of phase orders. Given the significant reduction in exploration time and other advantages of this approach, we believe that it is suitable for a wide range of compiler users concerned with performance.

Keywords: GPU · Phase ordering · Optimization

1 Introduction

Compilers optimize a function/program by applying a set of analysis and transformation operations over a representation of its source code (see, e.g., [2]). Those operations are implemented in compiler passes, each typically implementing a well delimited operation with a specific purpose, such as unrolling loops.

The set of compiler passes considered, and the order in which they are executed, can have a measurable impact in the quality of the final solution, for one

© Springer Nature Switzerland AG 2019
G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 494–505, 2019.
https://doi.org/10.1007/978-3-030-10549-5_39

or more given metrics of interest in the context of software compilation targeting Central Processing Units (CPUs) [16] and hardware compilation targeting Field-Programmable Gate Arrays (FPGAs) [10]. On Graphics Processing Units (GPUs), using this technique can also yield considerable improvements (e.g., up to 5× when targeting a NVIDIA Pascal-based GPU [15]).

The problem of finding orders of compiler passes (also called *compiler sequences* or *phase orders*) that result in better optimization (e.g., vs –O3) of a given function/program, for a given target and objective metric, is known as the *phase ordering* problem. The number of compiler passes available in current compilers is high and increasing (e.g., LLVM 3.3 has 157 passes, LLVM 3.9 has 245 passes). In LLVM, compiler phase ordering is accessible through passing ordered lists of flags to the LLVM Optimizer command-line tool (*opt*). Although the user interface is simple, the amount of available compiler passes results in a too large number of combinations to try manually. Moreover, compiler passes can have complex interactions (positive or negative) with other passes depending on when in the compilation process they are executed and depending on static and/or dynamic features of the function/program being compiled. Due to these factors, phase ordering is generally considered a difficult problem.

GPUs are widely used in a number of heterogeneous systems, such as smartphones, personal computers and supercomputers. Therefore, the performance of these systems strongly depends on how effectively GPUs are used. Purini and Jain [16] previously developed an approach for fast phase ordering, and evaluated it on CPUs. They found that their approach produced binaries that were comparable to slower state-of-the-art alternatives. In this paper, we evaluate their kind of approach in the context of OpenCL kernel compilation for GPUs. Going one step further, we propose and experiment with the use of a simple heuristic to make the approach more effective in generating suitable compiled code.

The rest of the paper is organized as follows. Section 2 provides background, including the description of aspects we believe to be important for an approach to address in order for it to suit a large number of compiler users concerned with optimization. Section 3 presents a selection of work in the field of Design Space Exploration (DSE) of compiler phase ordering, including the DSE approach that we augment with an heuristic. The heuristic we propose and evaluate in this paper is described in Sect. 4. Section 5 describes our experimental setup, including the target GPU and the OpenCL kernels used in our experiments. The experimental results are presented in Sect. 6. Finally, concluding remarks about the work presented in this paper are presented in Sect. 7.

2 Background and Motivation

This section presents what we believe are the important qualities a phase ordering specialization approach must have in order for it to suit a large number of compiler users and use cases.

2.1 Concerns Related with Phase Ordering

Given how difficult it can be to manually derive effective compiler phase orders, multiple automatic approaches have been proposed (see, e.g., [10,15,16]). Most approaches presented in the state-of-the-art, given a new program/function, generate new (i.e., previously untested) sequences for evaluation. Using these sequences without strenuous validation can result in a number of unwanted scenarios. These include the premature halting of the compiler execution, broken compiled code, or even the generation of compiled code that is functionally different than it should, which is arguably the worst possible outcome as it can be difficult to detect. Moreover, a number of the approaches suffer from an unacceptably high DSE overhead and/or they sacrifice too much in terms of the quality of the solutions.

Assuring Functional Correctness. It is known that even production compilers have bugs. Eide and Regehr evaluated thirteen production-quality C compilers and, for each, were able to find cases where incorrect code to access volatile variables was generated [9]. Iterative approaches for automatic phase ordering, such as the ones based on genetic algorithms or simulated annealing, typically rely on the generation and evaluation of a large number of compiler sequences (e.g., hundreds, thousands) during DSE to achieve considerable improvements in the compiled code in relation to an already optimized baseline (e.g., produced using the most aggressive optimization level). The number of sequences that can be generated by these iterative approaches is very large, so naturally most of them were not previously validated by the compiler writers, and expecting them to be exhaustively tested is not realistic. Therefore, compiler bugs are often exposed by these iterative methods. When custom compiler pass sequences are used, there is an high risk of side effects caused by bugs in any given compiler pass not previously detected by the battery of tests performed by the compiler developers. Even the validation of individual compiler passes is often incomplete. Zhao et al. [20] were able to create a formally verified version of the `mem2reg` pass, though they needed to rewrite the pass to do so and write $50,000$ lines of proof scripts and infrastructure. However, despite this significant effort, the formally verified pass was less optimized than the original non-verified pass.

Balancing Exploration Overhead and Solution Quality. DSE on top of standard compilation can add a considerable overhead. This is aggravated in cases where more than one execution per compiled version is required in order to cover multiple execution flows, which might be required for a more thorough validation of the generated compiled code. Exploration overhead might be significant for most compiler users, to be further aggravated if the execution time of the function/program is considerable (e.g., a function/program that even compiled with GCC/Clang –O3 takes 1 h to execute). In some cases, certain techniques can be used to reduce execution time of a function/program while still maintaining it representative of the original function/program (e.g., reduce number

of iterations of an outer loop) in a way that the same set of compiler knobs found for the code version modified in preparation for DSE can be used on the original version with comparable improvements. However, these techniques might not be straightforward to implement automatically. Either way, independently of the execution time for a given function/program, performing fewer compilations and executing the compiled code fewer times is preferable.

2.2 What Can Make an Approach Suitable to Most Compiler Users?

A considerable number of DSE approaches from the state-of-art are not suitable to most compiler users because of they require non-trivial validation by the final compiler user side and/or they require a large number of iterations to considerably improve most codes.

Figure 1 presents the different roles that actors in an approach of such type can take. To significantly lessen the requirement of validating (at the final user side) the functional correctness of the code compiled with the use of custom phase orders, we can evaluate only compiler sequences that have been previously demonstrated to work well on a set of representative functions/programs. Considerable efficiency (regarding number of compilations/evaluations) can be achieved by selecting a small, yet highly representative, set of these sequences.

Group	Compiler Developers	Phase Order Developers	Final Developers
Description	No changes needed (in LLVM's case) May also be the "phase order developers"	Develop target-aware phase orders with automated tools	Compile their programs Using the given phase orders and compilers
Result	Compiler	Set of phase orders	Final optimized program

Fig. 1. Roles in a type of approach that uses predefined custom phase orders.

3 Related Work

To the best of our knowledge, Cooper et al. [8] were the first to propose iterative compilation as a means to find phase orders to improve the quality of the compiled code with respect to a given metric. They used iterative compilation in the form of a Genetic Algorithm (GA) as a way to minimize the footprint of compiled code. Cooper et al. [7] explore compiler optimization phase ordering testing different randomized search algorithms based on genetic algorithms, hill climbers and randomized sampling. Almagor et al. [3] rely on GAs, hill climbers, and

greedy constructive algorithms to explore compiler phase ordering at program-level to a simulated SPARC processor. Nobre [13] presents results for the use of a approach that relies in simulated annealing to specialize compiler sequences in the context of software and hardware compilation. More recently, Nobre et al. [14] presented an approach based on sampling over a graph representing transitions between compiler passes, targeting the LEON3 microarchitecture.

Agakov et al. [1] present a methodology to reduce the number of evaluations of the program being compiled with iterative approaches. Models are generated taking into account program features and the shapes of compiler sequence spaces generated from iteratively evaluating a reference set of programs. These models are used to focus the iterative exploration for a new program, targeting the TI C6713 and AMD Au1500 embedded processors. Kulkarni and Cavazos [11] proposed an approach that formulates the phase ordering challenge as a Markov process where the current state of a function being optimized conforms to the Markov property (i.e., the current state must have all the information to decide what to do next). Instead of suggesting complete compiler sequences, these authors use a neural network to propose the next compiler pass based on current code features. Sher et al. [19] describe a compilation system that relies on evolutionary neural networks for phase ordering. Neural networks constructed with reinforcement learning output a set of probabilities of use for each compiler pass, which is then sampled to generate compiler sequences based on the input program/function features. Martins et al. [12] propose the use of a clustering method to reduce the exploration space in the context of compiler pass phase order exploration. Amir et al. [5] present an approach for compiler phase ordering that relies on predictive modeling, using dynamic features to suggest the next compiler phase to execute to maximize execution performance given the current status; and more recently, they presented MICOMP, an approach that performs phase ordering of the compiler passes in the sequences represented by LLVM optimization levels using sub-sequences and machine learning to predict the speedup of using combinations of subsequences [4].

Purini and Jain [16] presented and evaluated a type of approach that devises an universal set of compiler sequences that covers the program space of a reference set of programs. Given a new program, all and only sequences from that predefined set of sequences are evaluated. The authors demonstrated, using LLVM 3.3 to target a computer with an X86 CPU, that sets of compiler sequences that perform well on a set of reference functions can also be suitable to compile other functions to the same target. Purini and Jain demonstrated that these sets can be quite small (e.g., 10 in what they call the *Best-10* approach), while still being able to achieve considerable binary execution performance improvements. Comparing with other DSE approaches, this type of approach results in fast evaluation at the user/programmer-side as the set of representative sequences is small. This makes it feasible to perform an exhaustive offline validation, in a manner similar to that of the standard optimization levels (e.g., –O3). This is important because validation at the user-side would normally be expensive, involving the comparison the outputs of the compiled functions/programs with

sets of expected outputs for representative inputs, and even that might be insufficient as it may not cover all binary program paths.

The type of approach presented by Purini and Jain has a *training* phase, performed offline by the phase order developers; and an online phase, performed when a new program is compiled. In the training phase, other DSE methods are used to produce a large number of phase orders and corresponding metric improvements, of which a small set (K sequences) is selected.

In the online phase, only the sequences from the K set are evaluated. The validation of these sequences can be performed offline.

Purini and Jain presented multiple approaches to obtain the representative set of sequences, of which we opted to use the following:

1. Select the sequence that improves more kernels;
2. Select the sequence that combined with all previously selected sequences, maximizes the number of improved kernels. Use geomean as a tiebreaker;
3. Repeat 2. until K sequences ($K = 10$, in their paper) were selected.

4 Our Approach

Purini and Jain's approach [16] consists of generating a set of K sequences for each platform/compiler, and then using those sequences to compile any new programs/functions. However, if a user prefers to test fewer than K sequences, it is still possible to do so, by using the following algorithm:

1. Evaluate Seq. 1 (i.e., the first sequence extracted offline);
2. Evaluate the next sequence (i.e., by extraction order);
3. Repeat 2. until all K sequences were evaluated, the number of evaluations or time the user is willing to wait for is achieved, or the compiled code is sufficiently improved over baseline.

In this approach, the order of evaluation of the compiler sequences is always the same because, when the K set is constructed, each new sequence added is the one that best complements the sequences already obtained, so testing them in-order tends to yield better results, on average. We extend upon this type of approach by proposing the use of an heuristic to make the order of evaluation of the compiler sequences of the K set tuned to the code being compiled based on the impact of the previously evaluated sequences. This still circumvents the need to perform feature engineering and to classify code based on static and/or dynamic features. The only feature is the metric that one wants phase ordering to improve (e.g., performance). To the best of our knowledge, we are the first to evaluate this type of phase ordering approach.

4.1 Proposed Heuristic

Given a new code to compile, the end-user of the type of approach presented in [16] might not want to evaluate all K sequences. If less than K sequences are

to be evaluated at the end-user side, then selecting which of those sequences to evaluate is important. We will refer to the sequences from the K set that are to be evaluated for a given new code as the T set (where $1 \leq T \leq K$).

If no information about a given new code being compiled is taken into account, then giving preference to the evaluation of sequences added first to the K set during the training phase seems to be a good (though as we will see below, not optimal) approach. For instance, if the end-user only wants to evaluate 3 custom compiler phase orders, then Seqs. 1 to 3 would be evaluated in-order. In this scenario, the order in which the sequences from the T set are to be evaluated is not important, given all are evaluated. However, order of evaluation can be important in a scenario where it is not known from start at the user side how many custom compiler phase orders are to be evaluated. In the later case, first evaluating sequences from the T set with lower index likely yields better results.

Other than all K sequences having been evaluated (i.e., $T = K$), other possible stopping conditions for the process of evaluating sequences from the K set (from lower index to higher index) can be, for instance, reaching a given improvement over baseline or a maximum compilation/evaluation overhead.

It is important to note that, when evaluating less than K sequences, it is not always the best choice to evaluate the subsequent custom sequences form the K set that have lower index. The best subset of sequences from the K set to evaluate depends on the particular code being compiled.

We propose and evaluate an heuristic, that we formulated in order to allow achieving comparable improvements over baseline while requiring evaluating fewer custom phase orders from a given K set. Given information about the specific code being compiled, the end-result of using the heuristic is the evaluation of a particular sub-selection of the K sequences. Notice that an heuristic that is not suitable can result in losing efficiency over evaluating the sequences with lower index from the K set. The first compiler phase order is always evaluated, as it is by far the phase order that is most generic and other phase orders are selected to be part of K for their ability to improve upon the sequences with lower index.

The heuristic selects the next sequence from the K set to evaluate based on the impact of the custom sequence from the K set previously evaluated. The heuristic replaces point 2 of the process that selects the next sequence from the K set to evaluate (see Sect. 4). Instead of evaluating sequences by the order they were extracted from the initial set of pairs of sequences and fitness values, the algorithm chooses the sequence that is predicted to be the most likely to result in the highest speedup. It compares the result (e.g., speedup) of the last tested sequence with that of each training program to find which one had the closest result and verifies which of the untested K sequences produced the best results for that program.

4.2 Example

Consider the hypothetical example of Table 1, with $K = 4$ sequences. When compiling a code, the approach would first compile and measure the impact of using Seq. 1. Suppose that the speedup for this sequence is 5.5×. This means that the training code that is most similar to this case is CODE3, so the next sequence to test is Seq. 4 (as it has the highest speedup for CODE3 out of the 3 sequences that have not yet been tested).

Table 1. Hypothetical example of speedups for a set of sequences on a set of reference programs/functions.

Ref. code	Seq. 1	Seq. 2	Seq. 3	Seq. 4	Seq. 5
CODE1	2×	1×	2×	3×	0.2×
CODE2	4×	2×	0.1×	0.4×	0.1×
CODE3	6×	0.1×	0.1×	6×	1.2×

5 Experimental Setup

We used a workstation with an Intel Xeon E5-1650 v4 CPU, running at 3.6 GHz (4.0 GHz Turbo) and 64 GB of Quad-channel ECC DDR4 at 2133 MHz. For the experiments we relied on Ubuntu 16.04 64-bit with the NVIDIA CUDA 8.0 toolchain (released in Sept. 28, 2016) and the NVIDIA 378.13 Linux Display Driver (released in Feb. 14, 2017).

The GPU is an EVGA NVIDIA GeForce GTX 1070 graphics card (08G-P4-6276-KR) with a 1607/1797 MHz base/boost graphics clock (NVIDIA GP104 GPU) and 8 GB of 256 bit GDDR5 memory.

The kernel mode driver is set to keep the GPU initialized at all instances and the preferred performance mode is set to maximum performance to reduce the occurrence of extreme GPU and memory frequency variation during execution of the GPU kernels. All execution time metrics reported in this paper correspond to the average over 30 executions.

5.1 Kernels

In this paper we use the PolyBench/GPU benchmark suite [17] kernels. We selected this particular benchmark as it is freely available and thus contributes to making the results presented in this paper reproducible.

PolyBench/GPU is a collection of 15 kernels implemented for GPUs using CUDA, OpenCL, and HMPP; including convolution kernels (2DCONV, 3DCONV), linear algebra (2MM, 3MM, ATAX, BICG, GEMM, GESUMMV, GRAMSCH, MVT, SYR2K, SYRK), data-mining (CORR, COVAR), and stencil computations (FDTD-2D). We use the default datasets so that reproducibility of our results is more straightforward.

5.2 Compilation and Execution Flow with Specialized Phase Ordering

We use Clang compiler's OpenCL frontend with the `libclc` library to generate an LLVM IR representation of a given input OpenCL kernel. Then, we use the LLVM Optimizer tool (`opt`) to optimize the IR using a specific optimization strategy represented by a compiler phase order, and we link this optimized IR with the `libclc` OpenCL functions for our target using `llvm-link`. Finally, using Clang, we generate the PTX representation of the kernel from the bytecode resulting from the previous step, using the `nvptx64-nvidia-nvcl` target.

For specialized phase ordering, we use *offline compilation*, i.e., we compile the source code to PTX using Clang/LLVM and pass the resulting PTX code to the `clCreateProgramWithBinary` function.

5.3 Data Used for Devising a Small Set of Sequences

The OpenCL kernels from each of the benchmarks have been compiled/tested with a set of $10,000$ randomly generated compiler phase orders (the same set was used with all OpenCL codes) in the context of the work presented by Nobre et al. [15]. The data resulting from this strenuous evaluation is the input to the phase order extraction method used in the training phase. Only sequences that produce code that passes validation may be selected for the K set.

Each phase order is composed of 256 LLVM pass instances (can include repeated calls to the same pass) and the LLVM passes to consider for these sequences were selected from a list with all LLVM 3.9 passes except the ones that resulted in compilation and/or execution problems when used individually to compile the PolyBench/GPU OpenCL kernels.

6 Experimental Results

This section presents the results for the experiments performed to evaluate the efficiency of the proposed heuristic.

The evaluation of the approach was performed using 2-fold cross-validation. Randomly distributing the 15 PolyBench [17] kernels between two non-intersecting groups resulted in a group with `2DCONV`, `2MM`, `3MM`, `COVAR`, `GEMM`, `MVT` and `SYRK`; and another with `3DCONV`, `ATAX`, `BICG`, `CORR`, `FDTD-2D`, `GESUMMV`, `GRAMSCHM`, and `SYR2K`. The geometric mean metrics reported in this section consider the speedups obtained on all 15 codes.

The baseline used to calculate the speedups obtained when compiling the OpenCL codes using custom phase orders is the execution time of OpenCL versions generated by offline compilation using Clang/LLVM with `-O3`.

Note that the speedups reported in this paper with the use of custom phase orders would not be considerably higher or smaller if using other optimization levels or online compilation as baseline, because all standard optimization levels appear to be very similar for these kernels (see Nobre et al. [15]).

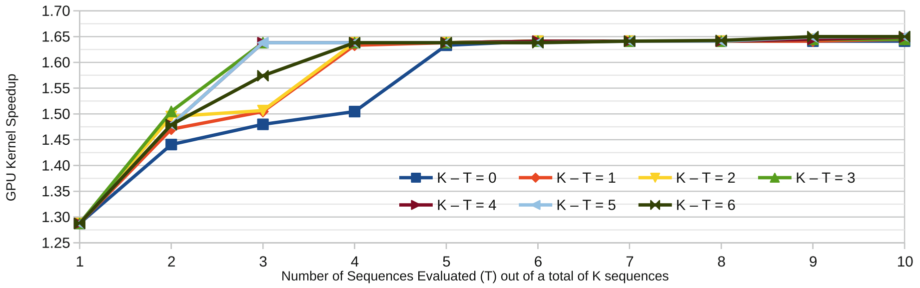

Fig. 2. Speedup achieved on tested kernels as a function of the number of evaluated sequences (T). Each line (K − T) represents a number of *excluded* sequences.

6.1 Impact of the Heuristic

Figure 2 depicts the speedups over baseline, considering different values (from 0 to 6) for the number of sequences excluded from evaluation, and different values for T (number of custom sequences from the K set that are evaluated).

As seen in Fig. 2, the ideal number of excluded sequences ($K - T$) is between 3 and 5. If too few sequences are excluded, then this method is not substantially different from Purini's approach (particularly when $K - T = 0$). If too many sequences are excluded, then for the same number of tested sequences (T), that means the compiler sequence selector must select from a higher number of available sequences of K. Since each additional generated sequence tends to be worse than all previous ones, that means that higher values of K imply worse average sequence quality.

For between 3 and 5 excluded sequences, a speedup of 1.638× is achieved with the evaluation of only 3 custom compiler sequences. Not relying on the heuristic ($K - T = 0$) required 5 evaluations to achieve compiled code with similar performance (1.633×). The ratio of improvement in efficiency with the use of the heuristic increases if considering even fewer evaluations. For the spectrum of values for the number of excluded sequences, evaluating only 2 custom compiler sequences results in compiled code that is similar in terms of performance with the compiled code obtained when relying on 4 evaluations without the heuristic.

Performing more that 3 evaluations (5 if not using the heuristic) does not result in significantly performance improvements of the compiled code. The geomean speedup obtained considering the use of the best individually found compiler sequence (per OpenCL code) from the 10,000 compiler sequences evaluated during the training phase is 1.653× (calculated using the same baseline, Clang/LLVM with –O3), making even the geomean speedup obtained with only 3 evaluations 0.99% of the former.

6.2 Generated GPU Code with Phase Ordering vs. Baseline

The performance increase with the use of phase ordering can be attributed to the use of different unroll factors, different memory loads (single combined

instruction vs. multiple instructions) and moving memory stores out of a loop. A more detailed analysis can be found in Nobre et al. [15].

7 Conclusions

This paper presented and evaluated in the context of improving the performance of code targeting a NVIDIA Pascal GPU, a heuristic-based extension to a previous approach evaluated in the context of specialized phase orders for CPUs. This type of approach has characteristics that make it potentially more suitable to a larger number of users and use-cases: fast and efficient exploration at the final user side and possibility of pre-validating all the sequences used by the final users. The proposed heuristic helps making evaluation at the user-side faster.

When considering very low numbers of compilations/evaluations, relying on the heuristic to accelerate iterative compilation resulted in achieving compiled GPU code of comparable binary execution performance while requiring significantly fewer compilations/executions. For instance, 2 evaluations of custom phase orders using the heuristic achieves performance similar to 4 evaluations without the heuristic, and 3 evaluations with the heuristic is comparable to 5 evaluations without the heuristic. Moreover, performing only the evaluation of 3 custom compiler sequences results in achieving a geometric mean speedup of 1.64×, while using the best sequence individually found for each code results in a performance improvement of 1.65×.

We are currently evaluating the impact of a number of modifications to the heuristic presented in this paper, such as considering features other than the speedups obtained with the use of a single compiler sequence (the previously evaluated sequence) when computing the distance metric, and using other distance metrics. Ongoing work also includes evaluating the approach with other GPUs, including GPUs from other vendors (e.g., AMD). In addition, we plan to evaluate with OpenCL kernels from other benchmarks with versions targeting GPUs, such as Rodinia [6] and SNU NPB Suite [18].

We believe that the use of the proposed heuristic can make optimization through specialization of compiler sequences accessible to an even larger number of compiler users.

Acknowledgements. This work was partially supported by the TEC4Growth project, "NORTE-01-0145-FEDER-000020", financed by the North Portugal Regional Operational Programme (NORTE 2020) under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). Reis acknowledges the support by FCT through PD/BD/105804/2014.

References

1. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: CGO 2006, pp. 295–305. IEEE Computer Society, Washington, DC (2006)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)

3. Almagor, L., et al.: Finding effective compilation sequences. In: LCTES 2004, pp. 231–239. ACM, New York (2004)
4. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J.: Micomp: mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. ACM TACO **14**(3), 29 (2017)
5. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C.: Predictive modeling methodology for compiler phase-ordering. In: PARMA-DITAM 2016, pp. 7–12. ACM, New York (2016)
6. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: IEEE IISWC, October 2009
7. Cooper, K.D., et al.: Exploring the structure of the space of compilation sequences using randomized search algorithms. J. Supercomput. **36**(2), 135–151 (2006)
8. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: LCTES 1999, pp. 1–9. ACM, New York (1999)
9. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT 2008, pp. 255–264. ACM, New York (2008)
10. Huang, Q., et al.: The effect of compiler optimizations on high-level synthesis-generated hardware. ACM TRETS **8**(3), 14:1–14:26 (2015)
11. Kulkarni, S., Cavazos, J.: Mitigating the compiler optimization phase-ordering problem using machine learning. In: OOPSLA 2012, pp. 147–162. ACM, New York (2012)
12. Martins, L.G.A., Nobre, R., Cardoso, J.M.P., Delbem, A.C.B., Marques, E.: Clustering-based selection for the exploration of compiler optimization sequences. ACM TACO **13**(1), 8:1–8:28 (2016)
13. Nobre, R.: Identifying sequences of optimizations for HW/SW compilation. In: FPL 2013, pp. 1–2, September 2013
14. Nobre, R., Martins, L.G.A., Cardoso, J.a.M.P.: A graph-based iterative compiler pass selection and phase ordering approach. In: LCTES 2016, pp. 21–30. ACM, New York (2016)
15. Nobre, R., Reis, L., Cardoso, J.M.P.: Impact of compiler phase ordering when targeting GPUs. In: Heras, D.B., Bougé, L. (eds.) Euro-Par 2017. LNCS, vol. 10659, pp. 427–438. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75178-8_35
16. Purini, S., Jain, L.: Finding good optimization sequences covering program space. ACM TACO **9**(4), 56:1–56:23 (2013)
17. Scott Grauer-Gray, L.N.P.: Polybench/GPU: Implementation of Polybench codes for GPU processing (2012). http://web.cs.ucla.edu/~pouchet/software/polybench/GPU/index.html
18. Seo, S., Jo, G., Lee, J.: Performance characterization of the NAS parallel benchmarks in OpenCL. In: IISWC 2011, pp. 137–148. IEEE Computer Society, Washington, DC (2011)
19. Sher, G., Martin, K., Dechev, D.: Preliminary results for neuroevolutionary optimization phase order generation for static compilation. In: ODES 2014, pp. 33–40. ACM, New York (2014)
20. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. SIGPLAN Not. **48**(6), 175–186 (2013)