Autotuning LLVM Optimization Passes for Matrix Multiplication in Rust

Emanuel Lima de Sousa, Pedro Bruel, Alfredo Goldman

[1]Institute of Mathematics and Statistics – University of São Paulo (IME USP)
Postal Code 05508-090 – São Paulo – SP – Brazil

emanuel.lima.sousa@usp.br, phrb@ime.usp.br, gold@ime.usp.br

Abstract. *The LLVM compiler framework transforms its Intermediate Representation (IR) to optimize code. These transformations are controlled by flags which interfere on metrics such as the execution time. Selecting flags to improve the execution time of a program is difficult, and requires expert knowledge. Autotuning methods can automate parts of this process and help understanding the underlying search spaces. This paper describes ongoing work, showing that LLVM flags can impact the execution time of a Rust matrix multiplication algorithm, and planning future autotuning experiments for flag selection.*

1. Introduction

Writing, porting, and optimizing code for High Performance Computing (HPC) architectures is challenging, and requires expert work and extensive knowledge of a specific hardware. Autotuning is one way to decrease the cost of optimizing a program, where we automatically explore a program configuration search space to minimize some target metric. In the ongoing work described in this paper, we intend to perform autotuning in the search space defined by compiler flags, to decrease the execution time of a matrix multiplication algorithm in the Rust language.

Compiler flag selection is a textbook autotuning application. Instead of relying on generic flag configurations, we can select flags with an autotuner, with significant improvements in certain cases. In previous work [4], for example, we show that autotuning CUDA compiler parameters achieves speedups of up to 3 times in comparison with the *-O2* flag. The LLVM compiler framework [6] uses flags to control optimizations that can improve performance. Rust is a systems programming language, and its compiler's front-end is implemented in LLVM. This paper shows that different LLVM passes impact the performance of a Rust matrix multiplication program, and describes how we will apply the Design of Experiments methodology (an autotuning technique) to identify and select the best flags for performance.

The paper is organized as follows. Section 2 provides a brief overview of LLVM and of the Rust programming language, and describes a search subspace of LLVM optimizations. Section 3 describes the LLVM pass selection search space Design of Experiments and other autotuning methods, and describes the subset of the search space we chose. Section 4 presents and discusses our preliminary and expected results. Finally, Section 5 describes future work.

2. LLVM Optimization Passes for Rust

Rust is a systems programming language that aims to provide memory safety and high performance [9]. Rust has been successfully applied on different domains, such as astro-

Table 1. Some of LLVM's IR optimization flags

Flag	*constprop*	*instcombine*	*aggressive-instcombine*	*jump-threading*	*lcssa*	*licm*	*loop-deletion*	*loop-extract*
Description	Identify and substitute constants	Replace redundant instructions	Replace redundant patterns	Remove condition double checking	Put loops in Single Static Assignment Form	Extract constant operations in loops	Remove dead loops	Extract loops to functions
Flag	*loop-reduce*	*loop-rotate*	*loop-simplify*	*loop-unroll*	*loop-unroll-and-jam*	*loop-unswitch*	*mem2reg*	*memcpyopt*
Description	Loop Strength Reduction	Rotate Loops	Canonicalize natural loops	Unroll loops	Unroll and Jam loops	Unswitch loops	Move memory to registers	MemCpy Optimization

physics [3] and operating system kernels [8]. The Rust compiler consists of a front-end for the LLVM compiler framework, and guarantees safety by enforcing data ownership and object lifetimes. Recent work approached the problem of tailoring the optimization of the LLVM IR for Rust [7].

The LLVM compiler framework provides an Intermediate Representation (IR) of the input code, which abstracts details both of the input language and of the target architecture [6]. To use LLVM's back-end to generate machine code for a new language, it is sufficient to write an LLVM front-end that generates IR code. Optimizations performed in the IR are called *passes*, and can improve the performance of the resulting binary. The *loop unrolling* pass, for example, transforms a loop so that it runs more than a single iteration before a condition check, with the potential to decrease the costs of condition checking and pointer arithmetic. Table 1 lists 16 LLVM passes that potentially impact performance. The next section describes the autotuning methods we intend to use to select LLVM passes to optimize Rust code.

3. Choosing LLVM Passes to Speedup Rust Programs

Although metrics such as compilation time and cache misses are important for program optimization, the objective in this paper is to find the flag selection which most improves the execution time of our matrix multiplication program, described in Section 4. There are 2^{16} possible combinations in the set of flags from Table 1, and it is impossible to explore such a large space exhaustively. To be able to find good flag selections within a reasonable amount of time, we will employ Design of Experiments (DoE), a parsimonious exploration approach.

Search spaces defined by autotuning problems are typically complex and riddled with local minima, which causes the performance of generic search heuristics to be usually equivalent to a uniform random sampling algorithm [2] that picks configurations at random and keeps the best one. Although we have applied search heuristics to compiler flag selection with good results in previous work [4], the analyses and explanations that could be produced from the collected data were limited, due to the uneven sampling those methods produce.

The DoE methodology provides parsimonious and statistically transparent methods to choose which experiments to run. *Screening* experiments, for example, assume a linear relationship between parameters and performance, and ignore interactions between parameters. The analysis of a screening experiment enables the identification of the most impactful parameters, or *main effects*, with minimal experimentation cost. We will perform screening experiments with LLVM passes for Rust in order to remove non-impactful passes from our search space, and then we will apply more flexible DoE methods, such as *optimal designs*, to detect interactions and construct a performance model. More details regarding the DoE approach can be found in our autotuning work on source-to-source transformation [5]. The next section will present preliminary experiments, evaluating the impact of some LLVM pass selections on the performance of matrix multiplication program in Rust, and discuss future experiments.

4. Impact of LLVM Flags on Matrix Multiplication in Rust

This work aims to use DoE methods to identify flags that impact performance, and to choose the set of flags that minimizes execution time. We show that 4 sets of flags impact the execution time of a matrix multiplication program in Rust, justifying and motivating further efforts to optimize flag selection. We implemented 4 program versions using different libraries, and allocating memory on the heap and on the stack. The code and all generated data are available on GitHub [1].

Measurements shown in Figure 1 were obtained for a version that allocates memory on the heap using the *Vec* native data structure, and multiplied randomly-generated square matrices of size 512. The comparison baseline was the "*release*" set of parameters in Rust's *cargo* build utility, which is the highest generic optimization level available. We also measured the performance of a configuration using no flags. We turned link-time optimization off on all experiments, to avoid its interference. We performed the experiments on an Intel Core i5 3230M with 8GB of RAM DDR3 1600MHz.

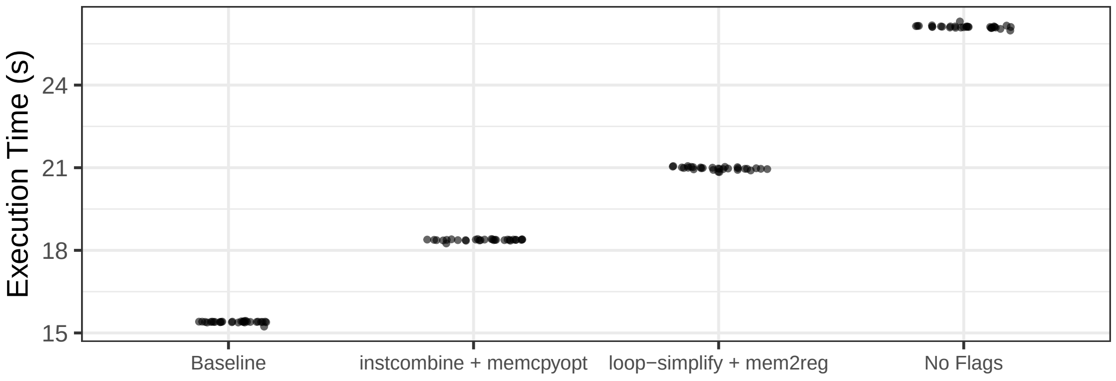

Figure 1. Effect of some flag combinations on performance of matrix multiplication in Rust. Each of the 30 repetitions for each experiment is shown spread over the corresponding experiment id.

The variance of execution times for each flag selection was extremely small, which means that further experiments can be made cheaper by doing less than 30 repetitions. We see a clear impact of flag selection on performance, and it is motivating to see that a simple flag selection can achieve performance approximately 20% slower than the highest generic optimization level available. Autotuning flag selection for Rust code seems promising, and moving forward we will construct a screening design for the 16 flags on Table 1, and we expect to reduce the flag search space by identifying flags that do not impact performance.

5. Conclusion

In this paper we discussed the autotuning problem in the context of compiler flags. We described the LLVM compiler framework, the Rust front-end, and how we can control the selection of optimization passes. We show that flag selections impact the performance of a Rust matrix multiplication program, highlighting that a simple flag selection achieves performance approximately 20% slower than the highest-level generic optimization. In future work we will run screening experiments to identify significant flags, and use optimal design to model the impact of passes on performance on a comprehensive Rust performance benchmark. We will also study the impact of matrix size in compiler flag selection. We expect to measure significant performance improvement in relation to the "*release*" parameters, and to identify which flags are responsible. We will compare performance improvements of flag selections made by our models with those made by a random sampling baseline, also measure the performance of our models in a comprehensive benchmark of Rust programs.

References

[1] Code and data on GitHub. `https://github.com/emanuellima1/matrix-multiply-test`. Accessed on March 14th, 2020.

[2] Prasanna Balaprakash, Stefan M Wild, and Paul D Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136–2145, 2011.

[3] Sergi Blanco-Cuaresma and Emeline Bolmont. What can the programming language Rust do for astrophysics? *Proceedings of the International Astronomical Union*, 12(S325):341–344, 2016.

[4] Pedro Bruel, Marcos Amarís, and Alfredo Goldman. Autotuning CUDA compiler parameters for heterogeneous applications using the OpenTuner framework. *Concurrency and Computation: Practice and Experience*, 29(22):e3973, 2017.

[5] Pedro Bruel, Steven Quinito Masnada, Brice Videau, Arnaud Legrand, Jean-Marc Vincent, and Alfredo Goldman. Autotuning under tight budget constraints: A transparent design of experiments approach. In *The 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019)*. IEEE/ACM, 2019.

[6] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[7] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P Lopes. Reconciling high-level optimizations and low-level code in LLVM. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[8] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[9] Nicholas D Matsakis and Felix S Klock. The Rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.