Course Project II

CS 526 — Advanced Compiler Construction
Spring Semester 2024

Due: End of Semester (04/29)

Goal

The primary goal of this project is to give you experience with designing and writing a major analysis or transformation in an optimizing compiler, by implementing (and preferably, extending) techniques in the literature. A secondary goal is to give you exposure to topics of current research interest in the compiler area – you are welcome (and highly encouraged) to include a research component in your project!

This is a team project. You must work in teams of 2 (unless you have a strong reason to work alone; in that case discuss with me). All members of a team get the same grade for the project. Because these are long team projects, they will be expected to meet a high standard for completeness, testing, and documentation. *Correctness, modularity, and code quality are more important than the amount of functionality you implement, so start simple and add functionality incrementally, testing thoroughly at each step.* If you plan to include a research-based component, you must discuss it with the instructor and get the approval before submitting the proposal. Overall, the research-based proposal should be directly related to the topic of the course (program analysis, program transformations), comparable in effort with the topics we discuss later in this document.

Development Tools You can use LLVM as the main general-purpose compilation platform to implement the project. You can also use contemporary compilers and programming languages such as XLA, TVM, MLIR, PyTorch compile or Halide to implement your project. We will cover some of the basics of these compilers towards the end of the course, however, you are welcome to use their infrastructures to implement your project. Please use an Ubuntu distribution for development and Docker images are preferred.

Testing and Evaluation. Pay special attention to testing and evaluating your implementation. For LLVM, you can use some of the real programs in `llvm/project/llvm-test/{MultiSource,External}`. Consider testing your analysis/optimization on the common benchmark suites, such as SPLASH (for CPU-intensive tasks), PARSEC (for multicore performance, also includes the updated version of SPLASH), Rodinia (for heterogeneous architectures), or MediaBench (for multimedia applications). You are encouraged to find and try public-domain C or C++ programs, e.g., GNU utilities (e.g., coreutils package, tar, grep, awk, sed), and larger applications often used in analyzing compilers, such as Apache (httpd), OpenSSH, Squid, MySQL, Povray, etc.

For tensor or domain-specific compilers, you can use the test or application suites for the respective compiler. For example, common application for tensor compilers involve neural network topologies such as Resnet family, VGG family, LSTMs, transformer models etc. Make sure all the programs you test are compilable and after your transformations, they are still producing the correct results.

Deliverables

Sunday, Mar 8, 11:59pm (Urbana Time Zone):

Short project proposal, 1-page PDF. Please submit using this form. `https://forms.gle/qxjf4avEmtnuSUK28`. You will be using the same form to submit different components of the project (by editing the original response). This proposal should include:

- a short description of the problem to be solved;
- a list of references;
- a short summary of what algorithm(s) you will implement (if they are taken from existing papers), *or* a short summary of the state of the art, and how you want to improve on this.

- a break-down of the work into a short list of tasks, *credible* (but tentative) completion dates and how the tasks will be divided between the two team members.

Your list of tasks should include two explicit tasks for testing: one for creating and testing with small unit tests, and another for setting up and testing with larger programs. Don't underestimate these tasks: they can each take one person a week or more, just to do a minimally adequate job. Moreover, *each of these two tasks should be shared by both team members*, i.e., both write unit tests and both set up and run real programs.

April 9, 11:59pm: **30% of the grade**

The progress report, 1-page PDF. By this stage you should have a reasonably complete suite of tests (including unit tests and medium and large programs). You should also have *partial working code for a subset of proposed functionality* that has been tested on all those test cases. The functionality can be small, but it should be solid. Describe what you have accomplished, including any relevant preliminary results for programs that work. 30% of the overall grade is reserved for your progress accomplished during these first $\tilde{4}$ weeks after sending me your project proposal. **I will schedule meetings with each group during the Week of April 14th to go over the progress.**

April 25: Project Presentation **20% of the grade**

Each group will present their work in one slot. I will later follow up with the exact schedule and the format.

April 29, 11:59pm: **50% of the grade**

Final (well documented) working code with tests (10%), experimental results (20%), and project report (20%). The test suite and applications should be properly documented. The code you submit should include your source code, an executable that can be run on a Ubuntu virtual machine, as well as *all* the input programs you used for your tests. Include any modified versions of the compiler source files. The report should explain where the source code, executables, and test programs are, and how to run your code. The report format is described below.

Final Report

Your final project report should be submitted using the same submission link in PDF form. It should be maximum 5 pages (10 pt font, single-column article format or two column ACM format), not including References and Appendix. Refer to Latex style files in conferences like PLDI, ISCA.

The report should include:

1. The problem statement and motivation (1/3 page).
2. Brief summary of existing work in the literature, with citations (1/2 page).
3. High-level overview of your algorithm or design (1 page).
4. Implementation details (1-1.5 pages):
   (a) Prior analyses or transformations required by your code.
   (b) Major code components (passes, data structures, and functions).
   (c) Testing strategy and status: unit tests, small source programs, larger source programs,
5. Experimental results: (1-2 pages): Choose results appropriate for your project.
6. References.
7. Appendix:
   - A description of where to find your source code, executable, and input programs, and how to run the executable for example inputs.

- An Extended Example (as long as necessary): Use part of one of the benchmarks (or design your example based on one of the benchmarks) to illustrate what your code does. Choose the example carefully to highlight the major technical capabilities and limitations. You can use more than one example if needed, but no more than absolutely necessary.

Working in Teams

Some things to keep in mind when working in a team for this project:

- *All members of a team will receive the same grade.* You are responsible for monitoring each other's progress. Discuss any potential problems between yourselves first to try to resolve them. If that doesn't work, bring me into the discussion.
- Plan your tasks so that you can make substantial progress independently.
- At the same time, plan to integrate your pieces in phases, at least 3-4 times during the course of the project. Integrating everything all at once for the final experiments may raise too many difficult problems, when you don't have the time to tackle them.
- Use *Pair Programming* for key stages of the project, including designing details of all the interfaces between the key components, integrating your pieces, and tracking down difficult bugs. If you are not familiar with Pair Programming, see `http://en.wikipedia.org/wiki/Pair_programming`. Whether or not you are familiar with it, see `http://www.wikihow.com/Pair-Program` for advice on how to go about it effectively.
- Be sure to divide up some of the key phases of the project: writing the unit tests; setting up larger programs for testing (do a few each); designing the overall code organization and major interfaces; running the experiments; writing the final report.

Suggested Projects Involving Known Techniques in the Literature

Following is a list of possible project ideas. These topics are given mainly as a reference of the scope of the project. For more suggestions and inspiration, talk to me about some interesting ongoing work connecting compilers/PL and machine learning or see recent proceedings of the PLDI, OOPSLA, and CGO conferences.

1. **Auto-vectorization through Superword Level Parallelism for SIMD Instruction Sets.**

   Generate vector code for short SIMD instruction sets (AVX / SSE* / Neon) using an algorithm that looks for isomorphic instructions within a basic block.

   References: Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, PLDI '00, pages 145–156, Vancouver, Canada, 2000.

2. **Solver-aided Vectorization**

   goSLP is an Integer Linear Programming based vectorization algorithm. It finds optimal packing strategies for forming vector instructions. You can implement the stock algorithm or you can extend goSLP. Some suggestions include extending goSLP for packing more than 2 instructions at a time and finding a better ILP encoding.

   References: Charith Mendis and Saman Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. Proc. ACM Program. Lang. 2, OOPSLA, Article 110 (November 2018), 28 pages.

3. **Revectorization**

Revec is a compiler pass written in LLVM that revectorizes vector code written in previous generation instructions to use newer generation instructions. Currently, Revec uses a heuristic algorithm to select such opportunities. Students can extend Revec to use better more principled solver-aided vectorization algorithms.

References: Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: program rejuvenation through revectorization. In Proceedings of the 28th International Conference on Compiler Construction (CC 2019). Association for Computing Machinery, New York, NY, USA, 29–41.

4. **Optimizations for Temporal Graph Neural Networks**

   TGLite is a programming framework written in Python that optimizes temporal graph neural networks. Currently, TGLite supports only offline training and one extension the students can do is to extend TGLite to support online or streaming training. Ask me for the paper and the details, if you are interested in this project.

   References: Yufeng Wang and Charith Mendis. TGLite: A Lightweight Programming Framework for Continuous-Time Temporal Graph Neural Networks. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024 (To Appear)

5. **Dynamic Tensor Rematerialization**

   Tensor rematerialization is important when generating code targeting machines with limited memory. You can implement the algorithm suggested in this paper and hopefully extend it.

   References: Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, Zachary Tatlock, Dynamic Tensor Rematerialization. ICLR 2021

6. **Tensor Graph Algebraic Rewrites**

   TASO uses algebraic rewrites on the tensor computation graph to optimize it. You can implement TASO and extend its rewrite system to cover more expressive tensor IRs such as XLA High Level Operators (HLO).

   References: Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 47–62.

7. **Compiling Ragged Tensors**

   Compiling tensor programs with non-rectangular shapes is important when dealing with inputs with irregular shapes. You can implement the technique mentioned in the following paper and hopefully extend it.

   References: Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry, The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding, MLSys 2022.

8. **Partial Redundancy Elimination via Lazy Code Motion**

   This is an elegant iterative bit-vector dataflow algorithm for PRE. The implementation should use the improvements to PRE described by Briggs and Cooper, which also describes how to perform PRE effectively starting with code in SSA form.

   References:

   (a) J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," In *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992.

   (b) Preston Briggs and Keith D. Cooper, "Effective partial redundancy elimination," In *Proc. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

9. **An Array Dependence Analysis Algorithm**

   Implement an array dependence analysis pass by choosing an algorithm or a combination of algorithms from the literature. Most of the implementation effort is in extracting the input information for each subscript pair (and loops bounds), and this has already been done for you in LLVM. Because of that, you should aim to implement at least two powerful tests, plus two simple ones. One or more simple tests such as the simple SIV tests can be used to handle the most common cases fast, and to fall back on a more powerful test when the simple test fails. Some possible choices for the powerful test are:

- The Delta Test: Goff, Kennedy and Tseng [PLDI 1991]
- The Range Test: Blume and Eigenmann [Supercomputing 1994]
- The Omega Test (use the Omega library): Pugh [CACM, Aug. 1992]

10. **Interprocedural Slicing Using Dependence Graphs**

    Program slicing identifies the subset of a program that affects a particular value (backward slicing), or the subset that is affected by that value (forward slicing). The technique was originally described by Mark Weiser, and is an important technique in advanced programming environments, program verification, performance modeling, and other problems.

    The first paper below describes an algorithm for computing interprocedural slices of whole programs, using a program dependence graph. The second paper replaces a key step of the algorithm with a much faster version.

    LLVM already includes all the infrastrucuture you need to build a simple, intraprocedural dependence graph: (a) SSA dataflow edges; (b) May-alias information for pairs of memory operations; and (c) control dependence information (via dominance frontiers for the reverse CFG).

    References

    (a) S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, 12(1), pp. 26–60, January 1990.

    (b) An important improvement to part of the algorithm is described in:
    T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up Slicing," *SIGSOFT '94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dec. 1994. ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), pp. 11-20.

11. **Automatic Parallelization Via Decoupled Software Pipelining**

    A state-of-the-art algorithm for automatic parallelization of complex loops is Decoupled Software Pipelining, described in the papers below. Implement either the original algorithm (PACT 2004) or the improved one (CGO 2008) in LLVM, using the existing SSA, memory- and control-dependence analyses.

    References:

    (a) "'Decoupled software pipelining with the synchronization array," Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. PACT 2004.

    (b) "Performance Scalability of Decoupled Software Pipelining," Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. ACM Transactions on Architecture and Code Optimization (TACO), 5(2), Aug. 2008.

    (c) "Parallel-Stage Decoupled Software Pipelining," Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David I. August. CGO 2008.

12. **Memory optimizations for GPUs.**

    Implement two optimizations for OpenCL – *memory coalescing* and *memory prefetching* (a). Use the NVPTX back end in LLVM to compile and run on nVidia GPUs.

    References:

    (a) Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In Proceedings of the *2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2010, pages 86–97, New York, USA, 2010.