Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Fine-tuning LLVM optimizations

Bc. Tomáš Drbota

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

June 24, 2018

Acknowledgements

TODO

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on June 24, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Drbota, Tomáš. *Fine-tuning LLVM optimizations.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

TODO

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

Modern compilers provide a wide range of different optimizations in an effort to increase the performance of the resulting program. Many of them already offer ways to customize the amount of optimization - for example -O flags in GCC, compiler arguments (e.g. -finline-functions in Clang) or in-line attributes (e.g. #pragma unroll), but the overall granularity of these options is low.

The purpose of this work is to investigate options for further customizing the execution of transformation passes in an optimizing compiler and comparing the results to automatically optimized code. The intent is to ensure there are ways for the user to be able to specifically declare which transformations should or should not be used in a given scope. Implementation is done using LLVM and its C/C++ frontend, Clang.

Keywords llvm, clang, compiler, transformation, optimization

Contents

List of Figures

Introduction

A large part of modern compilers is composed of various optimizations meant to either make the resulting program run faster, be smaller in size, or a compromise between these two. The problem is - how do you decide which optimizations to use, how, and in which order? Some optimizations may depend on others, some might be mutually exclusive, and some might even change semantics of the program itself in order to achieve faster processing speed. All these considerations have to be made by someone - in most cases the compiler itself.

But in specific scenarios - for example certain architectures or hardware, and even on specific problems, the compiler may not be able to make an optimal decision by itself. Furthermore, certain optimizations may alter the semantics of the program, which could be an unwanted side effect and might lead to unexpected results during execution.

In such cases, it's up to the programmer to specify which optimizations should be run, optionally along with detailed arguments for them. The problem is, there is currently no universal way to specify most of them in the code itself. There are exceptions, such as `#pragma loop` attributes (in C/C++ code) to specify loop unrolling options, but currently, the only way to truly fine-tune all the optimizations is to use the built-in `opt` tool, which only works on a module level, and depending on your needs, its usage can be unnecessarily verbose.

The purpose of this work is to investigate possibilities of extending this functionality into the code itself, and to provide more granularity in its application, i.e. allow the programmer to specify its scope, instead of having to apply it to the entire module.

Thus, the first part of the thesis focuses on introducing the framework itself, namely LLVM and Clang. I will describe how the compiler currently works and what kind of optimizations it is capable of, optionally if there already exists any way to influence them.

The second part will then expand upon this base, and will focus on de-

signing and implementing a general approach to "hint" the compiler what optimizations it should run, in which scope, and with what parameters.

After these improvements are implemented, I'm going to benchmark my code with these new customized optimizations, and compare the results to what the compiler would produce on its own.

Part I

Background

Chapter 1

LLVM architecture

LLVM [1], formerly known as *Low-Level Virtual Machine* is "a collection of modular and reusable compiler and toolchain technologies" [2]. Originally intended as a compiler infrastructure well-suited for modern programming languages, it has since grown into an umbrella term for a decently large group of sub-projects, intended for building everything related to compilers, that is frontends, backends, optimizers, Just-In-Time compilers etc. Some of these sub-projects will be discussed later in the chapter.

Today, LLVM is a popular choice for TODO

This work will be using LLVM and its toolset exclusively.

1.1 Structure

Written in C++, at its core LLVM uses a classical three phase compilation process [3], popular for static compilers. These three phases are namely the frontend, optimizer, and backend.

The frontend is responsible for parsing and validating the source code, generally producing some sort of intermediate form. The optimizer then takes this form, and does various transformations to improve the code's running time, and is mostly independent of the target. Finally, the backend generates actual machine code specific to the architecture.

The main benefit of this design is that a programmer writing source code doesn't need to worry about architecture specific assembly code or optimizations, and one code should work the same on every platform supported by the backend.

1.2 Intermediate Representation

As I have mentioned in the previous section, all three phases during compilation work with some sort of intermediate code. In LLVM this internal assembly

language is called *Intermediate Representation* [4] (*LLVM IR* for short, or just *IR*), and it's arguably one of the most important parts of LLVM.

LLVM IR is a well specified SSA (Static Single Assignment) based representation that provides type safety and various low-level operations. It is the only interface to LLVM's optimizer and is used throughout all phases of LLVM's compilation process.

There are three different forms of LLVM IR, all of which are equivalent:

- in-memory compiler IR
- on-disk bitcode representation (e.g. for JIT)
- a human readable assembly representation

The ability to have human readable IR is going to be extremely helpful, as it allows the developer to write or edit arbitrary IR and compile it to an executable file. This is difficult to do with other compilers, e.g. GCC's *Gimple* only allows printing its intermediate form, but you cannot trivially edit and compile it to a functioning program.

I am not going to go into large detail here regarding the exact structure of a well-formed IR module, but having an overview is going to be necessary for understanding the implementation.

1.2.1 Module

An LLVM program is composed of one or more modules, which represent a translation unit of the input files. Modules consist of functions, global variables and symbol table entries. These module files can be further merged together (along with symbol resolution etc.) using the LLVM linker, but that's not required.

Incidentally, the module level is where most of the current optimizations run on. That is, if you want to run a certain transformation pass on a function, the easiest solution is to use the `opt` tool and run it on the whole module.

1.2.2 Function

A function is normally a callable piece of code that returns some value, and in IR they are no different. Similarly to C, functions in IR consist of a definition and a declaration.

Function definition uses the `define` keyword, and specifies a number of parameters. These, among others, include visibility style, calling convention, parameter list, return type, a function name, optional function attributes, metadata, and a list of basic blocks.

Declarations are a bit more lightweight. They use the `declare` keyword, and share many parameters with function definitions, though they miss the

actual bulk of the function, that is things such as basic blocks, attributes or metadata.

1.2.2.1 Attributes

Function attributes are used to describe additional information about a function. They are part of the definition, but not the declaration. Attributes are part of the function itself, not its type, therefore different functions can be of the same type, but have different attributes.

Attributes will be used extensively throughout this work, as it is the most convenient way to communicate function-wide information to the optimizer. For example LLVM currently uses attributes to tell whether a function should be inlined or not.

To use a function attribute in the IR, it is generally a keyword (or a list of keywords, space separated) placed after the function type specified.

1.2.3 Basic Block

Basic blocks are the foundation of functions, forming the Control Flow Graph (CFG) of the containing function.

Each basic block contains a list of instructions, ending with a terminator function (e.g. branch or return). At the top of a basic block there can optionally be a label, giving this basic block a symbol table entry. If this label is not provided, it's automatically assigned one.

A special case of a basic block is the first basic block of any given function. It is immediately executed upon entrance into the function, and it may not have any predecessors (i.e. branches into this basic block from another one).

1.2.4 Instruction

Each basic block is composed of different instructions. Memory allocation, stores, loads, unary/binary instructions, terminators. It is not necessary to go into detail for every single one, but for this work, the important part about instructions are metadata.

1.2.4.1 Metadata

Every instruction can have different metadata attached to it, which will be used extensively in my implementation. Metadata is commonly used inside LLVM for hinting the code generator and optimizer, for example whether a loop should be unrolled and how much.

Metadata doesn't have a type, nor is it a value. There are two different types of metadata: strings and nodes.

Metadata strings are simple strings surrounded by double quotes, which can contain any (even non-printable) character. This is useful for setting various flags on an instruction.

Metadata nodes are similar, but other than a string they may also contain values. This is useful when you want to pass a specific parameter to an instruction, e.g. how many times a loop should be unrolled.

1.3 Clang

Clang [5] is part of the LLVM project and it's their frontend for C, C++, Objective-C and Objective-C++. It's integrated directly into the LLVM source code as a tool, making it easy to do and test any changes.

I will be using Clang to help produce customized IR to convey information to the optimizer and backend.

1.4 Compilation process

[insert picture here]

A standard compilation process can be described in a few steps.

First, the frontend (in our case Clang) parses the source code and produces an AST (Abstract Syntax Tree), which is then converted into LLVM IR.

This IR is then fed into the optimizer, which is a series of analysis and optimization passes. Depending on the optimization level, some, if not all of them, could be skipped. The result of this optimization is also IR, now hopefully running faster, taking less space, or both.

After we have the optimized IR, the code generator processes it. For each IR instruction or a sequence of instructions, it emits native machine code for the given target architecture, producing an executable program.

This is just the simplest procedure. There are more advanced things LLVM is capable of, such as link-time optimizations, which let LLVM optimize across e.g. file boundaries, by producing IR bitcode instead of native object files, letting LLVM aggregate them together and optimize a larger chunk.

Chapter 2

Optimizations using LLVM

2.1 Overview

Optimizations are a large part of LLVM. As mentioned earlier, their role is to take existing IR, and produce faster and/or smaller IR, before handing it over to the backend.

Optimizations in LLVM are implemented as what's called `Passes` [6]. LLVM passes are simply C++ classes that derive from the `Pass` class (usually one of its subclasses, e.g. `FunctionPass`).

LLVM has what's called a `Pass Manager`, which is a class responsible for scheduling and running various passes on a given unit. It is there where the optimization level and various flags come into play.

There are three distinct kinds of passes, `Analysis passes`, `Transformation passes` and `Utility passes`.

2.1.1 Analysis passes

An analysis pass is a pass that computes various information. This information may then be used for debugging, or it may be used to identify various patterns in a program, creating a higher level representation of it.

An example...

2.1.2 Transformation passes

Transformation passes do actual modifications on the IR. They may use existing Analysis passes, possibly invalidating them, requiring new analysis.

An example...

2.1.3 Utility passes

Utility passes are generally all passes that don't fit the former two categories.

An example...

2.2 Standard process

2.3 Polly

Polly [7] is an add-on optimizer for LLVM, providing high-level loop and data locality optimizations. It uses an abstract mathematical representation based on integer polyhedra for its analysis and optimizations.

This is very helpful for example in programs using frameworks such as OpenMP, which is commonly used for parallelizing code including loops.

2.4 Examples

Part II

Fine-tuning optimizations

Chapter 3

Design

3.1 Current state

Currently, LLVM provides a large amount of optimization passes, and the easiest way to influence them is to use an optimization level flag, e.g. `-O3`. This provides an easy high-level way to specify, which sets of optimizations you want, but it is currently extremely difficult to e.g. optimize on a certain level, but excluding a specific pass, or vice versa, run at a lower level of optimization and add new passes.

Alternatively, LLVM and Clang already have built-in support for certain actions, such as loop unrolling, but it is not currently flexible enough to trivially adapt to other optimizations, but it will serve as a good reference.

3.2 Clang frontend

The goal here is to create or use an existing way to feed optimization hints to the backend. Since I will be using C/C++ for the implementation, the easiest way is to use `#pragma` statements, as LLVM already has support for parsing them.

3.2.1 Pragma

TODO specify how exactly they'll work

3.3 LLVM backend

Considering most of the backend is C++ based and works with LLVM IR, that's where I will be doing most of the work.

I will be using IR attributes and metadata given by the frontend to influence the Pass Manager, and tailor the optimizations to my needs.

3.3.1 Pass Manager

TODO specify how exactly it will be changed

Chapter 4

Implementation

4.1 Backend

4.2 Frontend

Chapter 5

Evaluation

Conclusion

Bibliography

[1] Lattner, C. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu.`

[2] The LLVM Compiler Infrastructure Project. Available from: `http://llvm.org/`

[3] Lattner, C. The Design of LLVM. Available from: `http://www.aosabook.org/en/llvm.html`

[4] LLVM Language Reference Manual. Available from: `https://llvm.org/docs/LangRef.html`

[5] clang: a C language family frontend for LLVM. Available from: `https://clang.llvm.org/`

[6] LLVM's Analysis and Transform Passes. Available from: `https://llvm.org/docs/Passes.html`

[7] Polly - LLVM Framework for High-Level Loop and Data-Locality Optimizations. Available from: `https://polly.llvm.org/`

Appendix A

Acronyms

todo TODO

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format