Talk proposal for the LLVM Track

David Tweed, The Azimuth Project

LLVM is a modular system of compiler components with backends for most popular architectures. It is primarily designed as a compiler construction framework, but also provides facilities for Just-In-Time (JIT) compilation[1] *of code. Although a lot of interest has focused on the implementation of LLVM-based compilers for 'compiled ahead-of-time' (AOT) languages (eg, clang for C), one of the most exciting uses is to generate code on-the-fly, taking advantage of situation specific knowledge.*

This proposal is for a short (approximately 30 minute) talk that falls somewhere between a case study and a tutorial. While considering how LLVM can be used for various means of boosting performance in interactive exploratory programs, it will be covering the aspects at an intermediate level, not discussing API at the detail level of a full tutorial.

The Read-Evaluate-Print-Loop (REPL) is recognised as incredibly productive in many programming language paradigms, eg, Python, Prolog, Haskell, Smalltalk, logo, dc, etc. REPLs are particularly popular and effective for interactive number crunching and data exploration, such as Matlab, Scilab, Octave and Julia. Most of these employ some combination of calling 'canned routines' in low-level languages and JIT compilation in order to acheive acceptable performance. Julia is particularly interesting as it uses LLVM to JIT expressions within a REPL, but LLVM provides a simple and effective way for JIT compilation to be added to many more languages.

We follow a simple language crunching a large dataset and the mechanisms and performance changes from using LLVM's:

1. **Standard compiler tansformations.** LLVM provides many standard compiler optimizations that can be effective even for interactive code.
2. **Inlining.** In static compilation it is tricky to pick inlining opportunities that won't uselessly bloat the executable. JITing within a REPL, worthwhile cases are much more clear..
3. **Vectorization.** There is wide variety in the vector instructions on various CPU models. When JITing specific vectorization can be done.
4. **Special instructions.** Likewise other incredibly niche, target-specific instructions can be chosen when compiling at execution time.
5. **Inline compression.** For large datasets memory bandwidth is the first bottleneck. When JITing, compression instructions can be inserted.
6. **Language specific passes.** Languages are often designed to embody specific properties; these can easily be utilized via a new LLVM pass.
7. **Function specialisation.** Uers often write general code but only use specific cases. By cloning and specializing functions into different versions better performance can be acheived.

A particularly useful aspect of LLVM for implementing this is the way that virtually all modifications can be done at the level of LLVM-IR (which provides 'intrinsics' for the non arget-independent instructions). Thus portability is obtained for various architectures, an important facet of effective Open Source applications. The talk will show in moderate detail how the above steps are implemented via LLVM, and practical results will be demonstrated using the BEST dataset, demonstrating real-world speedups.

[1] JITing is a term encompassing many things. Here 'JIT compilation' is compiling a set of routines *immediately* before execution when their inputs are known. We don't discuss techniques for re-compiling profiled hotspots during execution due to LLVM's poor fit.