Programming Project 2 - Exploring Good Sequence of Program Transformations

Due date: May 14th, 2016 at midnight PST

Description: This project focuses builds on the many existing LLVM compilation transformations. As we discussed in class it is impossible to know without application-specific knowledge which program transformations are beneficial in terms of the various metrics of performance, namely, execution time, energy and code size. As such in this project you are asked to explore and empirical find which sequence of transformations lead to "good" results for the transformed code.

For this purpose and to keep this assignment feasible in a short amount of time, you are to explore a set of LLVM transformations already in the current installation. You are welcome to use more but at least we suggest that you experiment with the following ones:

1- Dead code elimination - dead argument elimination - dead type elimination - dead instruction elimination - dead store elimination - dead global elimination;
2- Constant propagation
3- Transformation of induction variables (and computations derived from them) into simpler forms
4- Promoting memory to register - memcpy optimization
5- Reassociate expressions

Based on these transformations and others that you decide to use, you need to explore that are the "best" sequence of transformations and for which codes. Because in general the best sequence is highly dependent on the code "characteristics", say amount of array references, nested loops, arithmetic vs. integer codes, etc., you are asked to also "find" good kernel codes (C functions) that exhibit "interesting" behavior. For instance, you need to think about which sequences are good for what kinds of codes and test your hypothesis by producing a sample code that in fact elicits that behavior.

You need to test your code and evaluate the generated transformed code against an un-optimized version comparing number of generated instructions, for various classes of instructions. We suggest that you compare the profile of the generated instructions in terms of arithmetic, control transfer and memory related operations. If you get ambitious you can actually profile the generated code and also report the number of executed instructions for selected inputs for each of the kernels.

Turn-In Instructions: You need to turn in your pass source files and corresponding auxiliary functions. Please make sure you isolate all you code in specific folders and include whatever Makefile the TA needs to have to compile and test your code. In the end you will be responsible for helping the TA to validate your input and output. Should this approach become infeasible to validate your programming project we might include a live demonstration with along side your computer at the convenience of the TA during her office hours. This can only be done with files you have provided on the project's deadline.