LLVM Analyses/Transform Passes and Optimizations

Written by Akilesh B and Phaneendra Babu

November 10, 2015

- Analysis and transform passes are part of LLVM Optimization features.
- Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program.
- A pass is a operation on a unit of IR.It could mutate the IR and also can compute something about IR.
- Units of IR can be functions, modules and basic blocks.
- All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass.
- Depending on how your pass works, you should inherit from the ModulePass, CallGraphSCCPass, FunctionPass or LoopPass, or RegionPass, or BasicBlockPass classes, which gives the system more information about what your pass does, and how it can be combined with other passes.

Analysis Passes:

- Analysis passes compute information that other passes can use or for debugging or program visualization purposes.
- Analysis passes compute some higher order information about the unit of IR without mutating it.
- The pass can be thought of as producing a result which can be queried. For suppose a DominatorTree is produced by running a analysis pass over a function.
- One analysis pass may use another analysis pass result to compute its result and it forms a dependency graph. Since IR is not mutated, results are valid.

Various options:

CFG :

- -dot-cfg : It prints the control flow graph into .dot file
- -dot-cfg-only : Prints the control flow without function bodies

CFG contains each function body as a separate state. If conditions, while loops, conditions in for loops, each one of these intermediate codes are represented by a separate state and based on their boolean value 2 links arrive from them. Also while-body and while-end are separate states. Similarly for-cond, for-body, for-inc are separate states.

Alias analysis:

- -aa-eval : Exhaustive Alias Analysis Precision Evaluator. Basically, for each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function. Alias evaluator reports percentage of "no", "may", "partial", "must", "mod", "ref" alias responses.
- -count-aa: Count Alias Analysis Query Responses. A pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds.

Dominator tree:

- -domtree: Dominator Tree Construction. This pass is a simple dominator construction algorithm for finding forward dominators.
- -dot-dom: Print dominance tree of function to "dot" file. This pass prints the dominator tree into a .dot graph.
- -dot-dom-only: Print dominance tree of function to “dot” file (with no function bodies).

Post dominance frontier:

- -domfrontier: Dominance Frontier Construction. This pass is a simple dominator construction algorithm for finding forward dominator frontiers.
- -dot-postdom: Print postdominance tree of function to “dot” file. This pass prints the post dominator tree into a .dot graph.
- -dot-postdom-only: Print postdominance tree of function to “dot” file (with no function bodies).

Call Graph:

- -dot-callgraph: Print Call Graph to “dot” file.
- -print-callgraph: Print a call graph.

SCC of all graphs:

- -print-callgraph-sccs: Print SCCs of the Call Graph.
- -print-cfg-sccs: Print SCCs of each function CFG.

Intervals:

- -intervals: Interval Partition Construction. This analysis calculates and represents the interval partition of a function, or a preexisting interval partition.In this way, the interval partition may be used to reduce a flow graph down to its degenerate single node interval partition (unless it is irreducible).

Regions:

- -regions: Detect single entry single exit regions. The RegionInfo pass detects single entry single exit regions in a function where a region is defined as any subgraph that is connected to the remaining graph at only two spots. Furthermore, an hierarchical region tree is built.

Transformation Passes:

- Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way.
- Transform passes can transform unit of IR in some way.
- They depened on analysis pass results and can preserve analysis pass results even when transforming IR.
- Tranformation passes cannot depend on other transformation passes.

Various transformations:

Dead code elimination: Dead code elimination is a compiler optimization to remove code which is unreachable or that does not affect the program results.

- -dce: Dead Code Elimination Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead.
- -die: Dead Instruction Elimination Dead instruction elimination performs a single pass over the function, removing instructions that are obviously dead.

Basic-Block vectorization: LLVM has two vectorizers: The Loop Vectorizer, which operates on Loops, and the Basic Block Vectorizer, which optimizes straight-line code. Unlike loop auto- vectorization, which operate on regions with non-trivial control flow, basic-block auto- vectorization operates within each basic block independently.

- -bb-vectorize: This pass combines instructions inside basic blocks to form vector instructions. It iterates over each basic block, attempting to pair compatible instructions, repeating this process until no additional pairs are selected for vectorization. When the outputs of some pair of compatible instructions are used as inputs by some other pair of compatible instructions, those pairs are part of a potential vectorization chain.

SimplifyCFG:

- -simplifycfg: Simplify the CFG. Performs dead code elimination and basic block merging. Specifically, removes basic blocks with no predecessors. Merges a basic block into its predecessor if there is only one and the predecessor only has one successor. Eliminates PHI nodes for basic blocks with a single predecessor. Eliminates a basic block that only contains an unconditional branch.

Unroll Loops:

- -loop-unroll: This pass implements a simple loop unroller. It works best when loops have been canonicalized by the indvars pass, allowing it to determine the trip counts of loops easily.

Loop Invariant Code Motion: Loop-invariant code consists of statements or expressions (in an imperative programming language) which can be moved outside the body of a loop without affecting the semantics of the program.

- -licm: This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible.

Sparse Conditional Constant Propogation: -sccp: Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise.
- Assumes BasicBlocks are dead unless proven otherwise.
- Proves values to be constant, and replaces them with constants.
- Proves conditional branches to be unconditional.

Tail Call Elimination:

- -tailcallelim: This file transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop.

Source code of analyses/transformation passes:

- Source codes can be found in llvm-3.7.0/lib/Analysis and llvm-3.7.0/lib/Transforms folders.
- Firstly for Analyser source codes.

CFG.cpp

- BasicBlock analysis. This family of functions performs analyses on basic blocks, and instructions contained within basic blocks.

Functions:

- FindFunctionBackedges - Analyze the specified function to find all of the loop backedges in the function and return them. The output is added to Result, as pairs of <from,to> edge info.
- GetSuccessorNumber - Search for the specified successor of a basic block and return its position in the terminator instruction's list of successors. It is an error to call this with a block that is not a successor.
- isCriticalEdge - Return true if the specified edge is a critical edge. Critical edges are edges from a block with multiple successors to a block with multiple predecessors.
- getOutermostLoop - LoopInfo contains a mapping from basic block to the innermost loop. Find the outermost loop in the loop nest that contains basic block.
- isPotentiallyReachableFromMany - When the stop block is unreachable, it's dominated from everywhere, regardless of whether there's a path between the two blocks.
- isPotentiallyReachable : The same block case is special because it's the only time we're looking within a single block to see which instruction comes first. Once we start looking at multiple blocks, the first instruction of the block is reachable, so we only need to determine reachability between whole blocks.

LoopPass.cpp

Functions:

- deleteLoopFromQueue - Delete loop from the loop queue and loop hierarchy (LoopInfo).// Notify passes that the loop is being deleted.
- insertLoop - Inset loop into loop nest (LoopInfo) and loop queue (LQ).

- redoLoop - Reoptimize this loop. LPPassManager will re-insert this loop into the queue. This allows LoopPass to change loop nest for the loop. This utility may send LPPassManager into infinite loops so use caution.
- cloneBasicBlockSimpleAnalysis : Invoke cloneBasicBlockAnalysis hook for all loop passes.
- deleteSimpleAnalysisValue - Invoke deleteAnalysisValue hook for all passes.
- addLoopIntoQueue : Recurse through all subloops and all loops into LQ.
- runOnFunction : run - Execute all of the passes scheduled for execution. Keep track of whether any of the passes modifies the function, and if so, return true.
- skipOptnoneFunction : OptimizeNone and transformation passes should skip it.

Transformer source codes:

BBVectorize.cpp

Functions:

- getPairPtrInfo : This determines the relative offset of two loads or stores, returning true if the offset could be determined to be some constant value.
- vectorizePairs : This function implements one vectorization iteration on the provided basic block. It returns true if the block is changed.
- isInstVectorizable : This function returns true if the provided instruction is capable of being fused into a vector instruction. This determination is based only on the type and other attributes of the instruction.
- areInstsCompatible : This function returns true if the two provided instructions are compatible (meaning that they can be fused into a vector instruction).
- getCandidatePairs : This function iterates over all instruction pairs in the provided basic block and collects all candidate pairs for vectorization.
- computePairsConnectedTo : Finds candidate pairs connected to the pair P = <PI, PJ>. This means that it looks for pairs such that both members have an input which is an output of PI or PJ.
- computeConnectedPairs : This function figures out which pairs are connected. Two pairs are connected if some output of the first pair forms an input to both members of the second pair.

- buildDepMap : This function builds a set of use tuples such that <A, B> is in the set if B is in the use dag of A. If B is in the use dag of A, then B depends on the output of A.
- pairsConflict: Returns true if an input to pair P is an output of pair Q and also an input of pair Q is an output of pair P. If this is the case, then these two pairs cannot be simultaneously fused.
- pairWillFormCycle: This function walks the use graph of current pairs to see if, starting from P, the walk returns to P.
- buildInitialDAGFor: This function builds the initial dag of connected pairs with the pair J at the root.
- pruneDAGFor : Given some initial dag, prune it by removing conflicting pairs (pairs that cannot be simultaneously chosen for vectorization).
- choosePairs : Given the list of candidate pairs, this function selects those that will be fused into vector instructions.

LICM.cpp

Functions:

- hoistRegion : Walk the specified region of the CFG (defined by all blocks dominated by the specified block, and that are in the current loop) in depth first order w.r.t the DominatorTree. This allows us to visit definitions before uses, allowing us to hoist a loop body in one pass without iteration.
- computeLICMSafetyInfo : Computes loop safety information, checks loop body & header for the possiblity of may throw exception.
- isTriviallyReplacablePHI : Returns true if a PHINode is a trivially replaceable with an Instruction.This is true when all incoming values are that instruction. This pattern occurs most often with LCSSA PHI nodes.
- isNotUsedInLoop : Return true if the only users of this instruction are outside of the loop. If this is true, we can sink the instruction to the exit blocks of the loop.
- sink : When an instruction is found to only be used outside of the loop, this function moves it to the exit blocks and patches up SSA form as needed. This method is guaranteed to remove the original instruction from its position, and may either delete it or move it to outside of the loop.
- cloneBasicBlockAnalysis : Simple Analysis hook. Clone alias set info.
- pointerInvalidatedByLoop : Return true if the body of this loop may store into the memory location pointed.

- inSubLoop : Little predicate that returns true if the specified basic block is in a subloop of the current one, not the current one itself.

Writing new transformation passes:

Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code. One of the main features of the LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets.

Pass to count the operators.

```
#define DEBUG_TYPE "op"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;

namespace{

// inherit form FunctionPass to access attributes of a function

class OperatorCount : public FunctionPass{

std::map<std::string,int > op; // this map is used to store opcode and integer numbers
char ID;
OperatorCount() : FunctionPass(ID){

virtual bool startHere(Function &F) {

cout << "Function " << F.getName() << '\n';

for (Function::iterator f = F.begin(),e = F.end(); f != e; ++f) {

for (BasicBlock::iterator i = f->begin(), e = f->end(); i != e; ++i) {

if(op.find(i->getOpcodeName())==op.end()){

op[i->getOpcodeName()] = 1;
}
else{
op[i->getOpcodeName()]+=1;
}

}
}

// Iterators are initalised
std::map <std::string> ,int> :: iterator it;
for(it=op.begin();it!=op.end();it++)
```

```
cout << i->first << " : " << i->second << "\n";

op.clear();

return false;
}
}
};
}
```

Effects of optimizations on performance:

LLVM has various optimization levels:

- -O0: It performs no optimization. The compilation happens fastest and it generates code which is maximum debuggable.
- -O1: It is based on O0 but some additional options are added like: -basicccg, -simplycfg, -inline-cost etc
- -O2: It is an ideal optimization which enables most optimization supported by LLVM.
- -Os: It is similar to O2 but enables more optimization, it can reduce code size.
- -Oz: It is similar to Os but code size is significantly reduced (whenever possible).
- -O3: It is similar to O2 but in an attempt to optimize too much, it can generate unnecessary and unwanted code.
- -O4: This performs link time optimization. Program optimization is done at link time.

Although O flag accepts any number from 0-9 only the above mentioned are performed. -On for any n>=3 is same as -O3.

Study of auto-vectorization in LLVM:

LLVM has two types of vectorizers:

- Loop vectorizer: It handles varieties of program having features like run-time checks of pointers, programs with loops with unknown trip count, reductions, if conversion pointer induction variables, reverse iterators, global structures alias analysis and vectorization of function calls. It performs optimization for these parts.
- SLP vectorizer: Combines similar independent instructions into vector instructions. Optimization is done on program features such as memory accesses, arithmetic operations, comparison operations.

Test case	With vectorization enabled (time in ms)	With vectorization disabled (time in ms)
1	79	402
2	76	310
3	73	91
4	369	393
5	143	728

- In test case 1,2 and 5 there is significant improvement in performance.
- Test case 1: Bitwise operations, Test case 2: Unknown loop bound and loop invariants, Test case 5: Loop with a boolean.
- There isn't much improvement in test cases 3 and 4.
- Test case 3: Data elements to be accessed parallely are not consecutive. Test case 4: Read accesses with unknown misalignment.

Loop optimizations using Polly:

Polly is a high level optimizing infrastructure for LLVM. The different regions in the code are identified and extracted by polly to generate optimized code. Programs having trivial loop structure are optimized to a greater extent than others which don't have one. It does optimization through traditional polyhedron techniques.

The static control flow is identified, analyzed and replaced with an optimized one ensuring that the control flow is itself not affected. Parallel loops are identified by polly and a code is generated which makes use of threads to mimic the existing parallelism. There is a measurable improvement in speed. Vectorization can help in optimizing programs involving nested loops.

The programs given in the question were run and following is the result:

Results:

Benchmark	Run time with polly	Run time without polly
mvt	0.15	0.16
seidel	0.27	0.27
2mm	3.21	17.43
3mm	4.41	28.29
gemm	1.46	9.38

There is a significant improvement in run time using polly.