Block Builder in VeGen

1 Intrinsic Builder

The Create method of the Intrinsic Builder class shows how to copy a code snippet from one function to another basic block in LLVM.

First, we can obtain the function from a module using the getFunction method. Then, we can get a basic block from the obtained function (a function consists of one or more basic blocks). We also need to get the arguments from the function. All these tasks can be done through the function's iterator API.

Second, we create a value-to-value map to copy some IR instructions in the basic blocks. IR instructions with use-def relations can form a structure similar to a DAG. Copying a DAG requires a map, and so does copying instructions, since we have to let the new instruction refer to the new value. For function arguments, we first check whether the actual arguments (called operands) can be bitcast to the function formal parameters. If they can, we create a bitcast and map the function argument to the operand, since the new instructions generated by the copy need to refer to the operand, not the original argument.

Third, we examine all the instructions in the basic block until we meet a return instruction. We clone an instruction, insert the new instruction into the new block, and use the RemapInstruction LLVM function to replace the values according to the vmap, which means replacing all the operands of the newly created instruction with the new values. If the instruction is a call, which must be a call to another intrinsic in VeGen, we declare a new function in the new module and modify the call instruction to call it. This step is needed because functions will not be cloned like normal values and cannot be in the vmap. Finally, we get the new return value from the vmap using the old return value.

2 Block Builder

2.1 Test

To test the block builder, a LLVM context is required. A LLVM context contains the type and constant uniquing tables, which are necessary for programmatically constructing an IR module.

2.2 Block and Condition

Blocks and conditions are closely related. A block is executed under a certain condition, and a condition can have a block (in a control flow graph) that is executed when the condition is met. We can freely decide which code is executed under which condition, as long as the condition is represented by a compiler-understandable predicate combined using the logical operators "and" and "or".

2.3 ConditionEmitter

The ConditionEmitter is responsible for emitting the actual LLVM IR expression that computes the condition. This expression is used in the BlockBuilder as the IR branch condition.

2.4 getBlockFor

This function takes a control condition as input and returns a basic block that belongs to a valid control flow graph (CFG) constructed based on the input condition. The function implements a recursive graph algorithm that traverses the control condition and recurs on the parent of the And condition and the CommonC of the Or condition. It uses two lookup tables, one for recording active conditions and the other for recording semi-active conditions.

2.4.1 Active Condition

2.4.2 Semi-Active Condition