VLoop in VeGen

1 Loop Related Files

There are three groups of files that are directly related to loops in VeGen: Vloop, LoopUnrolling, and UnrollFactor. Each group has a header file and a cpp file.

2 VLoop

2.1 Members

VLoop is built upon the original LLVM loop and contains a pointer to the original loop. It has references to two control conditions: one decides whether to execute the loop, and the other is the back edge condition, which is essentially the loop condition in C but contains a chain of conditions from the function entry.

VLoop also contains a reference to the VectorPackContext. We can see a design pattern here: an object often has references to its context and to other supportive data structures that belong to the context. Here, the context includes the GlobalDependenceAnalysis and the VLoopInfo (which is a friend class of VLoop). VLoop has bit vectors to store the instructions that depend on the loop and the instructions that are contained in the loop, as well as a reference to the subloops.

Some condition-related things are here. They are maps from phi nodes to mu nodes, from phi nodes to one-hot phis, and from phi nodes to lists of control conditions (Gated phis). An instruction has a guard value, which is the value that instructions outside of this loop should use. Also, there are conditions for each instruction.

2.2 Methods

2.2.1 haveIdenticalTripCounts

First, try to use the LLVM ScalarEvolution class to obtain the back edge taken count of loop 1 and loop 2. If both are computable and equal, the loop trip counts are identical.

If not, check the exit block of the two loops. Only in this case do we need to consider further: both have an exit block, and the terminators of the exit blocks are identical.

Rely on the ScalarEvolution framework to recognize the loop counter, which is an affine expression of the trip. If the affine expression is equivalent, then the two loops have identical trip counts.

2.2.2 isSafeToFuse

To fuse, the loops should be control-equivalent, in the same loop level, independent (two loops are independent if neither has any common instructions with the other's dependent instructions) and having the same trip counts. Moreover, their parents should be safe to fuse.

2.2.3 isSafeToCoIterate

Loops that execute under different control predicates or have different trip counts cannot be fused, but they can become co-iterate if they are independent of each other. The difference in control can be resolved in the loop body by using any control statement, and the difference in trip count is also a form of control that depends on the loop variable. Therefore, the only prerequisite left is the independence of the loop body.

2.2.4 fuse

Fuse is a method of VLoopInfo. First of all, we need to fuse loops in a top-down manner, that is, merge two loops only when their parents are merged. For two loops VL2 and VL1, we try to fuse VL2 into VL1.

It is worth noticing that in VeGen, there is no clear separation between transform pass and analyze pass. I think this causes many analysis results to go into data structures that hold IR, making context, analysis, and LLVM IR jam together.

For example, VLoop defined by VeGen contains not only a reference to LLVM loop and instructions, but also holds dependent analysis results as bit vectors for instructions in the loop and instructions the loop depends on. When fusing two loops, these analyses are updated with fusing. This again makes me think that LLVM project has a clear design.

After fusing, VL2 is added to the deleted loops of VLoopInfo, and erased from the common parent's subloops.

2.2.5 coiterate

VLoopInfo uses an EquivalenceClasses (union-find) ADT to store co-iterated loops.

Coiteration operates on the leaders of the EquivalenceClasses.

3 LoopUnrolling

Loop unrolling replicates a value in multiple iterations of a loop body. VeGen defines UnrolledValue as a data structure to track the value across iterations. It also implements its own version of llvm::UnrollLoop that maintains the unrolled values.

3.1 needToInsertPhisForLCSSA

LCSSA stands for Loop Closed Static Single Assignment. LCSSA ensures that every value defined inside a loop and used outside the loop is assigned to a PHI node at each exit block of the loop. This makes loop analysis and transformations simpler and more effective. If we see a loop as a special function (inlined), then LCSSA is putting the return values of the function loop into the exit block, so they can be read from there.

In VeGen, needToInsertPhisForLCSSA checks all the instructions outside the loop. If any outside loop instruction has an operand which is defined in a loop containing the loop we are checking, then this loop needs to insert phis for LCSSA. Note that basic blocks cannot nest, but loops can.

3.2 simplifyLoopAfterUnroll2

3.3 UnrollLoopWithVMap

3.3.1 LoopUnrollResult & UnrollLoopOptions

LLVM defines an enum class called LoopUnrollResult, which has three possible values: Unmodified, PartiallyUnrolled, and FullyUnrolled. Unmodified means that the loop was not modified by the unrolling process. VeGen will not unroll the loop in some situations, such as when the loop has no pre-header, when the loop has more than one latch block, when the loop is not safe to clone (loops with indirect branches cannot be cloned), or when the loop header has its address taken (there are any uses of this basic block other than direct branches, switches, etc. to it).

LLVM also defines a struct called UnrollLoopOptions, which contains several member variables that specify various options for unrolling loops. The following member variables are used by VeGen: TripCount, Count, PeelCount, TripMultiple, AllowRuntime, UnrollRemainder, ForgetAllSCEV, PreserveCondBr, and PreserveOnlyFirst. Their meanings are as follows:

- → Count: The number of times to unroll the loop. A value of 0 means to use the default heuristic.
- → TripCount: The number of iterations that the loop will execute. A value of 0 means unknown.
- → AllowRuntime: A flag that indicates whether to generate code that can handle loops with unknown trip counts at runtime.
- → PreserveCondBr: A flag that indicates whether to preserve the conditional branch of the loop latch.
- → PreserveOnlyFirst: A flag that indicates whether to preserve only the first loop exit and remove the others.
- → TripMultiple: The largest constant divisor of the trip count of the loop. The actual trip count is always a multiple of it. The trip multiple is useful for loop transformations such as unrolling, peeling, or vectorization, as it indicates how many iterations can be executed without affecting the loop semantics. For instance, if a loop has a trip multiple of 4, then it can be unrolled by a factor of 4 without changing the loop behavior. However, if the loop has a trip multiple of 1, then unrolling by any factor would require additional checks or padding to preserve the loop semantics. A value of 0 means unknown.
- → PeelCount: The number of iterations to peel off before unrolling the loop. A value of 0 means to use the default heuristic.
- → UnrollRemainder: A flag that indicates whether to unroll the remainder loop.
- → ForgetAllSCEV: A flag that indicates whether to forget all the SCEV information about the loop after unrolling. SCEV stands for Scalar Evolution, which is a way of analyzing the values of scalar expressions in loops.

3.3.2 Trip Count

If Count is equal to TripCount, then the loop control is eliminated altogether, which results in a FullyUnrolled loop.

3.3.3 Peel Loop

```
preheader:
  br label %header
header: ; Loop header
  %i = phi i64 [ 0, %preheader ], [ %next, %latch ]
  %p = getelementptr @A, 0, %i
  %a = load float* %p
latch: ; Loop latch
  %next = add i64 %i, 1
  %cmp = icmp slt %next, %N
```

```
br i1 %cmp, label %header, label %exit
```

A loop is a subset of nodes from the control-flow graph with the following properties:

- The induced subgraph is strongly connected (every node is reachable from all others).
- All edges from outside the subset into the subset point to the same node, called the Header.
- The loop is the maximum subset with these properties.

The PreHeader is a non-loop node that has an edge to the Header. It dominates the loop without itself being part of the loop. The Latch is a loop node that has an edge to the Header. A backedge is an edge from a Latch to the Header.

3.3.4 UnrollRuntimeLoopRemainder

4 UnrollFactor

4.1 computeUnrollFactor

4.2 unrollLoops

4.3 getSeeds

This function returns a vector of operand packs that can be used as seeds for vectorization. An operand pack is a set of instructions (LLVM Values) that have the same type and can be packed into a vector register. The function takes three parameters: Pkr is a reference to a Packer object, which is a class that performs vectorization on LLVM IR; DupToOrigLoopMap is a reference to a dense map that maps duplicated loops to their original loops and iteration numbers; UnrolledIterations is a reference to a dense map that maps unrolled instructions to their original instructions and iteration numbers.

The function does the following steps:

1. It checks if the ForwardSeeds flag is set, which indicates whether to use forward propagation to find seed operands. if not, it returns an empty vector.
2. It gets a reference to the loop info analysis of the packer, which provides information about the loops in the function.
3. It defines a lambda function named GetUnrollVector, which takes an instruction pointer as an argument and returns a vector of unsigned integers. The vector represents the unroll vector of the instruction, which is a sequence of iteration numbers for each loop level that the instruction belongs to. The lambda function does the following:
    - It initializes an empty vector X.
    - It gets the basic block and the loop that contain the instruction.
    - If the instruction is not in any loop, it returns X.
    - It pushes the unrolled iteration number of the instruction in the innermost loop to X. If the instruction is not unrolled, it pushes zero. This is done by using the UnrolledIterations map.

- It iterates over the loop and its parent loops, and pushes the unrolled iteration number of the loop to X. If the loop is not duplicated, it pushes zero. This is done by using the DupToOrigLoopMap map.

- It returns X.

4. It gets a pointer to the function that the packer is working on.

5. It declares a map named InstToDim, which maps an original instruction to the dimension of its unroll vector. It also declares a map named UnrolledInsts, which maps a pair of an original instruction and its unroll vector to the corresponding unrolled instruction.

6. It iterates over all the instructions in the function, and does the following for each instruction:

    - It checks if the instruction is unrolled by using the UnrooledIterations map. If not, it continues to the next instruction.

    - It gets the original instruction and the unroll vector of the current instruction by using the map and the lambda function.

    - It inserts the pair of the original instruction and the unroll vector, and the current instruction to the UnrolledInsts map.

    - It inserts the original instruction and the dimension of the unroll vector to the InstToDim map.

7. It gets the maximum vector width that the target can support for load and store instructions by using the target transform info analysis of the packer.

8. It creates a reverse post-order traversal object for the function, which allows iterating over the basic blocks in reverse post-order.

9. It gets a pointer to the vector pack context of the packer, which is a class that manages the operand packs and their canonical forms.

10. It declares a vector of const operand pack pointers named SeedOperands, which will store the result of the function.

11. It iterates over the basic blocks in reverse post-order, and does the following for each basic block:

    - It iterates over the instructions in the basic block in reverse order, and does the following for each instruction:

    - It gets the type of the instruction and checks if it is void or vector. If so, it continues to the next instruction.

    - It computes the maximum vector length that the instruction can be packed into by dividing the maximum vector width by the bit width of the instruction type.

    - It checks if the instruction is unrolled by using the UnrolledIterations map. It gets the original instruction and the unroll vector of the current instruction by using the map and the lambda function.

- It iterates over the dimensions of the unroll vector, and does the following for each dimension:

    i. It tries to find a chain of instructions along the dimension that can be packed together. A chain is a sequence of instructions that have the same original instruction and the same unroll vector except for the current dimension. The chain starts with the current instruction and grows by incrementing the value of the current dimension in the unroll vector and looking up the UnrolledInsts map. The chain stops when it reaches the maximum vector length or when there is no matching instruction in the map.

    ii. If the chain has more than one instruction and its size is a power of two, it creates an operand pack from the chain and inserts it to the SeedOperands vector. It also gets the canonical form of the operand pack from the vector pack context and inserts it to the SeedOperands vector. The canonical form is a unique representation of an operand pack that ignores the order and duplication of the instructions.

12. It returns the SeedOperands vector.