Implementing SPMD control flow in LLVM using reconverging CFGs

Vectorizing divergent control flow for SIMD applications

Fabian Wahlster
Technical University Munich Department of Informatics
fwahlster@outlook.com — 1 (000) 111 1111

Abstract

Compiling programs for an SPMD execution model, e.g. for GPUs or for whole program vectorization on CPUs, requires a transform from the thread-level input program into a vectorized wave-level program in which the values of the original threads are stored in corresponding lanes of vectors. The main challenge of this transform is handling divergent control flow, where threads take different paths through the original CFG. A common approach, which is currently taken by the AMDGPU backend in LLVM, is to first structurize the program as a simplification for subsequent steps.

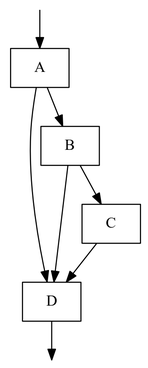
However, structurization is overly conservative. It can be avoided when control flow is uniform, i.e. not divergent. Even where control flow is divergent, structurization is often unnecessary. Moreover, LLVM's StructurizeCFG pass relies on region analysis, which limits the extent to which it can be evolved.

We propose a new approach to SPMD vectorization based on saying that a CFG is reconverging if for every divergent branch, one of the successors is a post-dominator. This property is weaker than structuredness, and we show that it can be achieved while preserving uniform branches and inserting fewer new basic blocks than structurization requires. It is also sufficient for code generation, because it guarantees that threads which "leave" a wave at divergent branches will be able to rejoin it later.

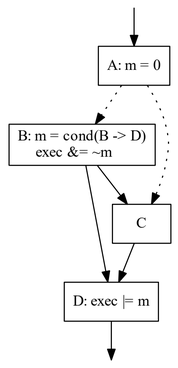
Reconverging control flow graphs

We argue that the structurization used in LLVM's **StructurizeCFG** region pass is too intrusive with respect to the input control flow. The weaker notion of *reconvergence* is sufficient for re-joining diverging threads required to generate wave-level code, while also handling uniform control flow properly.

Definition 1. *A control flow graph is reconverging if every non-uniform condition node (terminator T) has exactly two successors, one of which post-dominates it (primary successor).*



(a) Nested reconverging CFG.



(b) Nested CFG with incoming uniform edges.

Figure 1: Reconverging CFGs are suitable for lowering to wave-level control flow by inserting instructions for execution mask manipulation.

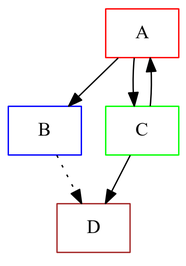
The simple definition of a *reconverging* CFG introduced in this work shows that it's powerful enough to effectively lower thread-level to wave-level control flow, solving the common problem of divergence management in code generation for SPMD and SIMT applications. Listing 1 shows the results of the lowering algorithm based on a reconverging input CFG.

Listing 1: Pseudo GCN ISA predication for control flow of Figure 1a

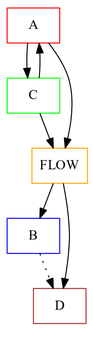
```
1   A:  // code for A
2       v_cmp_???       s[0:1], ...          // initialize re-join mask in m
3       s_andn2_b64     exec, exec, s[0:1]   // subtract re-join mask stored in s[0:1]
4       s_cbranch_execz D
5
6   B:  // code for B
7       v_cmp_???       vcc, ...             // condition for jumping to D
8       s_or_b64        s[0:1], vcc          // accumulate re-join mask in m
9       s_andn2_b64     exec, exec, vcc      // subtract re-join mask from execution mask
10      s_cbranch_execz D
11
12  C:  // code for C
13  D:  s_or_b64        exec, exec, s[0:1]   // add previously subtracted re-join masks
14      // code for D
```

Main Contributions

- Concise definition of a reconverging CFG
- Lowering algorithm exploiting the properties of such a CFG
- Algorithm transforming arbitrary and irreducible input CFGs to contain reconvergance points while also preserving uniform control-flow
- Evaluation of basic block ordering methods used as input of the CFG transformation



(a) Neither successor B or C post-dominate A.



(b) Flow block post-dominates A, hence becomes the primary successor.

Figure 2: CFG of Figure 2a inflicts Definition 1. CFG of Figure 2b is suitable for lowering to wave-level.

Algorithm

The algorithm for vectorizing divergent control flow using the properties of reconverging CFGs consists of the following steps:

1. Make sure a **unique sink** exists by adding a common virtual exit to be able to merge control flow from non-uniform branches that lead to different exit nodes.
2. Determine an **ordering** of the basic blocks based on the topology of the CFG. The correctness of the following transformation is not affected by the ordering created in this step, but the quality of the resulting CFG depends on it.
3. Transform the control flow to ensure existence of **reconvergence points** by adding **flow** blocks and rerouting edges accordingly.
4. Inject **predication operations** to lower the reconverging CFG to **wave-level**. Optionally keep scalar branch instructions to skip blocks that would be executed with an empty execution mask.

```
A:
    %cc_A = icmp eq i32 %in_A, 0
    br i1 %cc_A, label %B, label %C
B:
    br label %D
C:
    %cc_C = icmp eq i32 %in_C, 0
    br i1 %cc_C, label %A, label %D
D:
    ret void
```

(a) LLVM IR for CFG of Figure 2a.

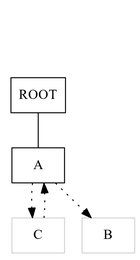
```
A:
    %cc_A = icmp eq i32 %in_A, 0
    br i1 %cc_A, label %FLOW0, label %C
B:
    br label %D
FLOW0:
    %0 = phi i1 [ true, %A ], [ false, %C ]
    br i1 %0, label %B, label %D
C:
    %cc_C = icmp eq i32 %in_C, 0
    br i1 %cc_C, label %A, label %FLOW0
D:
    ret void
```

(b) LLVM IR for CFG of Figure 2b.

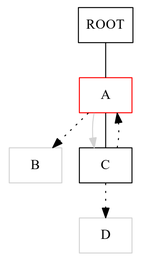
Figure 3: Phi instructions need to be insterted at *flow* blocks to maintain original control flow.

OpenTree structure

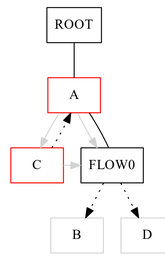
The transformation of input CFGs is executed on a bookkeeping structure called *OpenTree*, short OT. Open (dotted) edges in the OT are initialized with edges form the input CFG but maybe changed (rerouted) over the course of the algorithm. Solid light grey edges are closed. Visited nodes have a **black** frame, unvisited nodes light grey. **Solid** black edges symbolize OT parent-child edges. Divergent nodes are called armed (red) if one of the outgoing edges has already been closed.



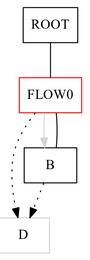
(a) Attach A to root node. Discover C and B.



(b) Attach C as child to A and close the edge. A is divergent and becomes armed. Subforest rooted at A and C contains multiple unvisited outgoing edges.



(c) Subforest of A and C is rerouted through new *flow* block. Unvisited nodes B and D become open edges the inserted block.



(d) B is attached to its lowest predecessor FLOW0, leaving only D which will collapse the OT to contain only the root node.

Figure 4: Maintaining OT structure for the control flow graph of Figure 2a. Processing basic blocks in order: $A \rightarrow C \rightarrow B \rightarrow D$.

Input orderings

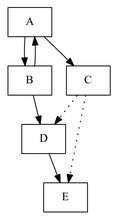
The input ordering of basic blocks affects the quality of generated control flow as it changes when and how the critical edges are detected and rerouted while processing the nodes of the OpenTree.

DF Conventional depth-first traversal

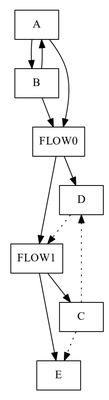
DFPD Depth-first obeying post-dominance

BF Conventional Breadth-first traversal

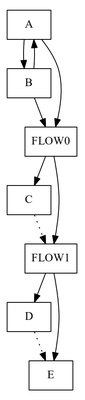
RPOT Reverse post order traversal



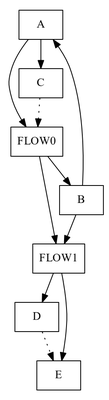
(a) CFG with uniform terminator C.



(b) DF, DFPD and BFPD.



(c) BF.



(d) RPOT.

Figure 5: Influence of input orderings on preserving uniform control flow when reconverging CFG of Figure (a)).