LLVM Example: Instruction_Move

작성일 2019-04-40

작성자 강석원

Instruction의 위치 이동

Instruction의 실행 순서는 일반적으로 LLVM IR 내에서 Instruction의 위치와 매핑됩니다.[1]

따라서 Instruction의 실행 순서를 바꾸기 위해서는, LLVM IR 내에서 Instruction의 위치를 조정하면 됩니다.

이러한 동작을 도와주는 API는 llvm::Instruction 클래스의 moveBefore(), moveAfter() 가 있습니다.

```
108   /// Unlink this instruction from its current basic block and insert it into
109   /// the basic block that MovePos lives in, right before MovePos.
110   void moveBefore(Instruction *MovePos);
111
112   /// Unlink this instruction and insert into BB before I.
113   ///
114   /// \pre I is a valid iterator into BB.
115   void moveBefore(BasicBlock &BB, SymbolTableList<Instruction>::iterator I);
116
117   /// Unlink this instruction from its current basic block and insert it into
118   /// the basic block that MovePos lives in, right after MovePos.
119   void moveAfter(Instruction *MovePos);
120
```

/usr/local/llvm/include/llvm/IR/Instruction.h

각 API를 이용해 현재 가지고 있는 llvm::Instruction 객체가 대표하는 Instruction의 순서를 특정한 Instruction 전/후로 이동시킬 수 있습니다.

---

[1] 물론 1) 타겟 하드웨어 Instruction 레벨로 내려가고 최적화 됨에 따라 2) 추가적인 IR-level 최적화에 따라 3) 타겟 하드웨어의 Out-of-Order 실행 유닛에 따라 LLVM IR과 변경될 수 있지만, 1), 3)은 변경할 수 없거나 매우 어려운 부분이고, 2)의 경우, llc를 이용한 llvm static compiler를 사용하면 추가적인 IR-level 최적화 없이 LLVM IR을 타겟 하드웨어 Instruction으로 변경할 수 있습니다.

Instruction_Move

LLVM 예제 Instruction_Move는 이러한 일련의 작업을 하는 OpenCL Kernel to NVPTX 컴파일러로써, 0번째 Load Instruction을 moveAfter()를 사용해 바로 앞으로(즉 실행 순서를 늦추는 방향으로), 1번째 Load Instruction을 moveBefore()를 사용해 바로 뒤로(즉 실행 순서를 앞당기는 방향으로) 변경합니다.[2]

```
; ModuleID = '__kernel.temp.bc'
source_filename = "2DConvolution.cl"
target datalayout = "e-i64:64-i128:128-v16:16-v32:32-n16:32:64"
target triple = "nvptx64-nvidia-nvcl-"

; Function Attrs: convergent noinline nounwind
define spir_kernel void @Convolution2D_kernel(float addrspace(1)* nocapture readonly, floa
g_base_type !7 !kernel_arg_type_qual !8 {
  %5 = tail call i64 @get_global_id(i32 0) #3
  %6 = trunc i64 %5 to i32
  %7 = tail call i64 @get_global_id(i32 1) #3
  %8 = trunc i64 %7 to i32
  %9 = add nsw i32 %2, -1
  %10 = icmp sgt i32 %9, %8
  br i1 %10, label %11, label %72

; <label>:11:                                     ; preds = %4
  %12 = add nsw i32 %3, -1
  %13 = icmp sgt i32 %12, %6
  %14 = icmp sgt i32 %8, 0
  %15 = and i1 %13, %14
  %16 = icmp sgt i32 %6, 0
  %17 = and i1 %16, %15
  br i1 %17, label %18, label %72

; <label>:18:                                     ; preds = %11
  %19 = add nsw i32 %8, -1
  %20 = mul nsw i32 %19, %3
  %21 = add nsw i32 %6, -1
  %22 = add nsw i32 %20, %21
  %23 = sext i32 %22 to i64
  %24 = getelementptr inbounds float, float addrspace(1)* %0, i64 %23
  %25 = load float, float addrspace(1)* %24, align 4, !tbaa !9
  %26 = add nsw i32 %20, %6
  %27 = sext i32 %26 to i64
  %28 = getelementptr inbounds float, float addrspace(1)* %0, i64 %27
  %29 = load float, float addrspace(1)* %28, align 4, !tbaa !9
  %30 = fmul float %29, 5.000000e-01
  %31 = tail call float @llvm.fmuladd.f32(float %25, float 0x3FC99999A0000000, float %30)
  %32 = add nsw i32 %6, 1
```

[2] 여기서 의미하는 순서는 전체 llvm::Module에서의 순서를 의미함

Dependency와 Error

하지만 이렇게 Instruction을 이동할 때에는 Instruction간의 Dependency를 고려해야 합니다.

0번째 Instruction을 앞 쪽으로 옮기는 것은 (원래의 IR에서) %25의 명령어와 %26의 명령어 간의 Dependency가 없어서 잘 처리되었지만, 1번재 Instruction을 뒤쪽으로 옮기는 것은 (원래의 IR에서) %28의 결과 값을 %29에서 사용하기 때문에 오류가 발생합니다.

이 오류는 메모리 상에서 IR 단계의 처리나, IR을 human-readable IR(.ll)로 출력할 때는 발생하지 않지만, 이를 bitcode(.bc)로 컴파일하려고 할 때 발생합니다.

```
25 ; <label>:18:                                     ; preds = %11
26   %19 = add nsw i32 %8, -1
27   %20 = mul nsw i32 %19, %3
28   %21 = add nsw i32 %6, -1
29   %22 = add nsw i32 %20, %21
30   %23 = sext i32 %22 to i64
31   %24 = getelementptr inbounds float, float addrspace(1)* %0, i64 %23
32   %25 = add nsw i32 %20, %6
33   %26 = load float, float addrspace(1)* %24, align 4, !tbaa !9
34   %27 = sext i32 %25 to i64
35   %28 = getelementptr inbounds float, float addrspace(1)* %0, i64 %27
36   %29 = load float, float addrspace(1)* %28, align 4, !tbaa !9
37   %30 = fmul float %29, 5.000000e-01
38   %31 = tail call float @llvm.fmuladd.f32(float %26, float 0x3FC99999A0000000, float %30)
39   %32 = add nsw i32 %6, 1
```

0번째 Instruction이 바로 앞으로 옮긴 결과.

```
25 ; <label>:18:                                     ; preds = %11
26   %19 = add nsw i32 %8, -1
27   %20 = mul nsw i32 %19, %3
28   %21 = add nsw i32 %6, -1
29   %22 = add nsw i32 %20, %21
30   %23 = sext i32 %22 to i64
31   %24 = getelementptr inbounds float, float addrspace(1)* %0, i64 %23
32   %25 = add nsw i32 %20, %6
33   %26 = load float, float addrspace(1)* %24, align 4, !tbaa !9
34   %27 = sext i32 %25 to i64
35   %28 = load float, float addrspace(1)* %29, align 4, !tbaa !9
36   %29 = getelementptr inbounds float, float addrspace(1)* %0, i64 %27
37   %30 = fmul float %28, 5.000000e-01
38   %31 = tail call float @llvm.fmuladd.f32(float %26, float 0x3FC99999A0000000, float %30)
39   %32 = add nsw i32 %6, 1
40   %33 = add nsw i32 %20, %32
```

1번째 Instruction이 바로 뒤로 옮긴 결과. 나중에 수행되는 명령어(%29)의 결과를 그 이전에 수행되는 명령어(%28)이 사용하고 있다.

```
Instruction does not dominate all uses!
  %29 = getelementptr inbounds float, float addrspace(1)* %0, i64 %27
  %28 = load float, float addrspace(1)* %29, align 4, !tbaa !9
```

Instruction간의 Dependency가 무너졌을 때 Instruction does not dominate all users 오류나, Segmentation Fault가 발생한다.

User

이러한 Dependency는 llvm::User 클래스와 llvm::Use 클래스로부터 알 수 있습니다.

User클래스는 Operand가 있는 Value들이 가지는 Class로, Instruction 클래스의 부모 클래스 입니다.

```
44 class Instruction : public User,
45                      public ilist_node_with_parent<Instruction, BasicBlock> {
46   BasicBlock *Parent;
47   DebugLoc DbgLoc;                         // 'dbg' Metadata cache.
48
49   enum {
50     /// This is a bit stored in the SubClassData field which indicates whether
51     /// this instruction has metadata attached to it or not.
52     HasMetadataBit = 1 << 15
53   };
54
```

/usr/local/llvm/include/llvm/IR/Instruction.h

이러한 User 클래스에는 getNumOperands(), getOperand() 같은 API들이 있어, 이를 사용하여 현재 Value에 사용되는 Operand들을 알 수 있습니다.

만약 어떤 Instruction을 보다 뒤로(실행순서를 이전으로) 옮기고 싶은데, Instruction의 Operand 중 일부가 Instruction인 경우, 이러한 Operand Instruction도 같이 뒤로 옮겨져야 합니다. 동시에, Instruction과 Operand Instruction의 상대적인 순서는 지켜져야 합니다.

또한 최상위 클래스인 llvm::Value에서는 UseList가 존재하여 현재 Value가 사용되는 부분, 즉 Operand의 입장에서 Operand들 사용하는 Instruction들을 알 수 있습니다.

```
73 class Value {
74   // The least-significant bit of the first word of Value *must* be zero:
75   //   http://www.llvm.org/docs/ProgrammersManual.html#the-waymarking-algo
   rithm
76   Type *VTy;
77   Use *UseList;
78
79   friend class ValueAsMetadata; // Allow access to IsUsedByMD.
80   friend class ValueHandleBase;
81
82   const unsigned char SubclassID;   // Subclass identifier (for isa/dyn_ca
   st)
83   unsigned char HasValueHandle : 1; // Has a ValueHandle pointing to this?
84
```

/usr/local/llvm/include/llvm/IR/Value.h

주로 use_begin(), use_end()를 사용하여 접근을 주로 사용합니다.

참고자료

[1] "llvm/IR/Instruction.h", "llvm/IR/User.h", "llvm/IR/Value.h"

[2] https://llvm.org/doxygen/classllvm_1_1Instruction.html[3]

[3] http://llvm.org/doxygen/classllvm_1_1User.html

[4] https://llvm.org/doxygen/classllvm_1_1Value.html

[3] [2], [3], [4] 사용하는 LLVM Version이 다름에 유의