Compiler Background 2

Live-variable Analysis

Learning Objectives

1. Explain the concept of Liveness Analysis
2. Explore the application of Liveness Analysis in register allocation
3. Identify data-flow equations and explain their role in analysis

Live-Variable (Liveness) Analysis

1. Liveness analysis helps determine which variables are live (in use) at various program points
2. Usage: register allocation
    - Register is allocated only for live variables, ensuring registers are allocated only to live variables

Data-Flow Equations

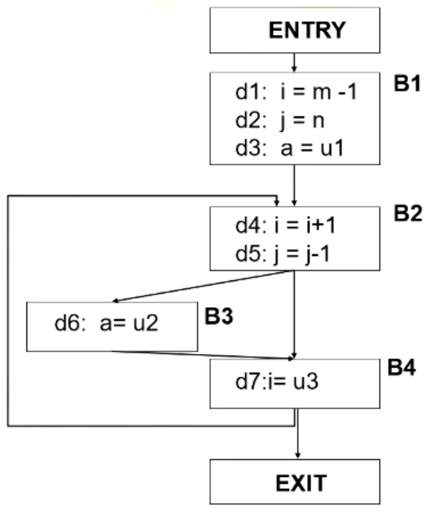
1. defB: set of variables defined in block B before any use
2. useB: Set of variables whose values may be used in block B before any definition
3. IN[EXIT] = NULL: boundary condition
4. IN[B} = useB U (OUT[B] - defB)
5. OUT[B] = U IN[S] where S is a successor of B
6. Analysis is done backward (opposite to the control flow)

Algorithm

1. Iterative process

```
IN[EXIT] = NULL;
for (each basic block B other than EXIT) IN[B] = NULL;
while (changes to any IN occur) {
    for (each basic block B other than EXIT) {
        OUT[B] = U IN[S] where S is a successor of B
        IN[B] = useB U (OUT[B] - defB)
    }
}
```

Example of Live-Variable Analysis



$$def_{B1} = \{i, j, a\} \quad use_{B1} = \{m, n, u1\}$$
$$def_{B2} = \{i, j\} \quad use_{B2} = \{i, j\}$$
$$def_{B3} = \{a\} \quad use_{B3} = \{u2\}$$
$$def_{B4} = \{i\} \quad use_{B4} = \{u3\}$$

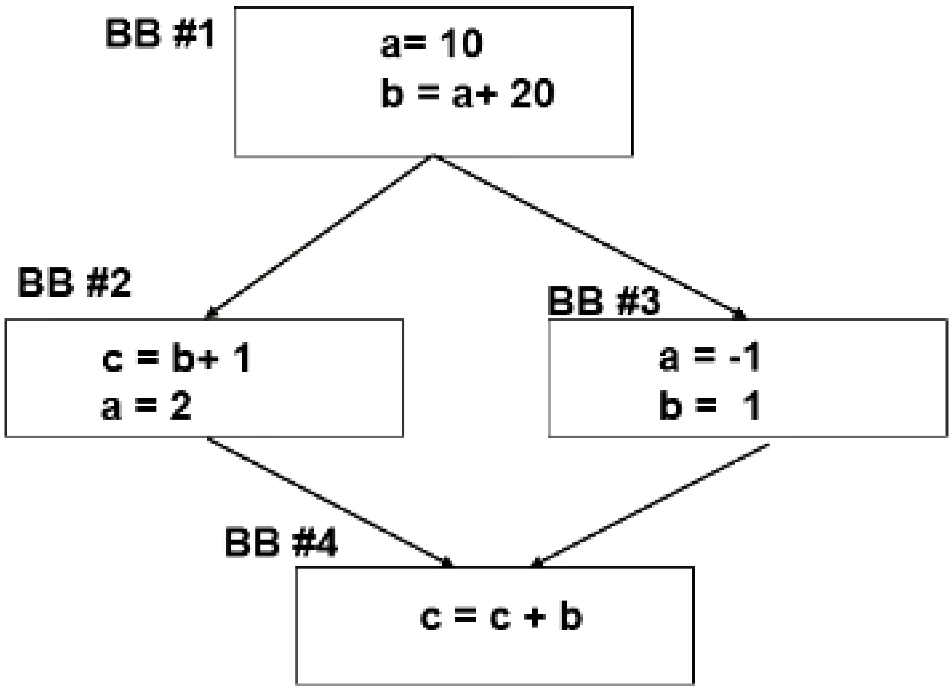
$$\text{OUT[B]} = \cup_{S\ a\ successor of\ B}\ \text{IN[S]}$$
$$\text{IN[B]} = use_B \cup (\text{OUT[B]} - def_B)$$

BB	First Pass	Second Pass
OUT[ENTRY]	m, n, u1,u2,u3	m, n, u1,u2,u3
IN[B1]	m,n,u1,u2,u3	m,n,u1,u2,u3
OUT[B1]	i,j,u2,u3	i,j, u2,u3
IN[B2]	i,j,u2,u3	i,j,u2,u3
OUT[B2]	u2,u3	j,u2,u3
IN[B3]	u2,u3	j,u2,u3
OUT[B3]	u3	j,u2,u3
IN[B4]	u3	j,u2, u3
OUT[B4]	Ø	i, j, u2, u3
IN[EXIT]	Ø	

Example of Live-Variable Analysis

Register Allocations and Live-Variable Analysis

1. a is dead after BB1
2. Register for a can be reused after BB1
3. b is still live at BB2
   - If b is dead at BB2, the register for b can also be reused



Register Allocations

Register Allocations

1. Only live variables need to have registers
2. What if there aren't enough registers available?
3. Register spill/fill operations to a stack
4. Values that won't be used for a while are moved to the stack
5. PTX assumes infinite number of registers, so stack operations are not explicitly shown

Summary of Data-Flow Analysis

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward	backward
Transfer function $f_b(x)$	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$
Boundary Condition	OUT[ENTRY] = $\emptyset$	IN[EXIT] = $\emptyset$
Meet Operation ($\wedge$)	$\cup$	$\cup$
Equations	OUT[B] = f_b(IN[B]) IN[B] = $\wedge$ out[pred(b)]	IN[B] = f_b(OUT[B]) OUT[B] = $\wedge$ in[succ(b)]
Initialize	OUT[B] = $\emptyset$	IN[B] = $\emptyset$

Summary of Data-Flow Analysis

Summary

1. Data-flow analysis for live-variable analysis
2. Reaching definition analysis uses forward analysis but live-variable analysis uses backward analysis

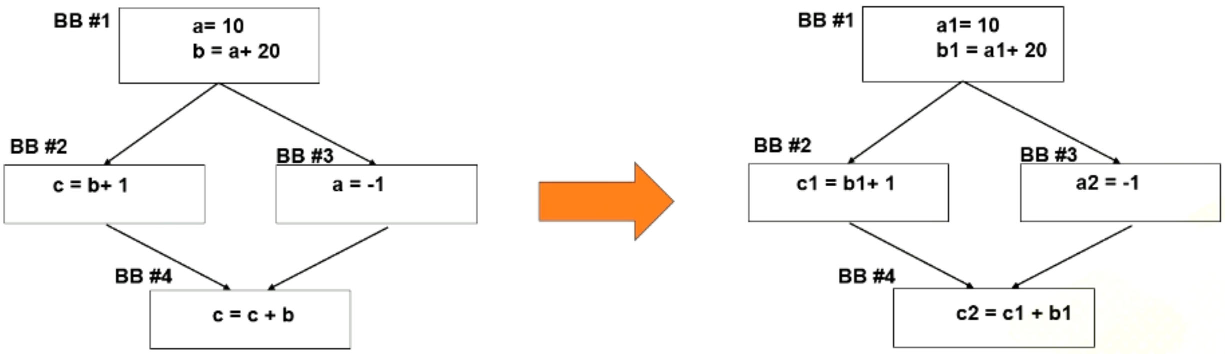
SSA

Learning Objectives

1. Explain the concept of SSA (static single-assignment) form
2. Explore the basics of converting code to SSA form
3. Explain how to merge values from different paths

SSA

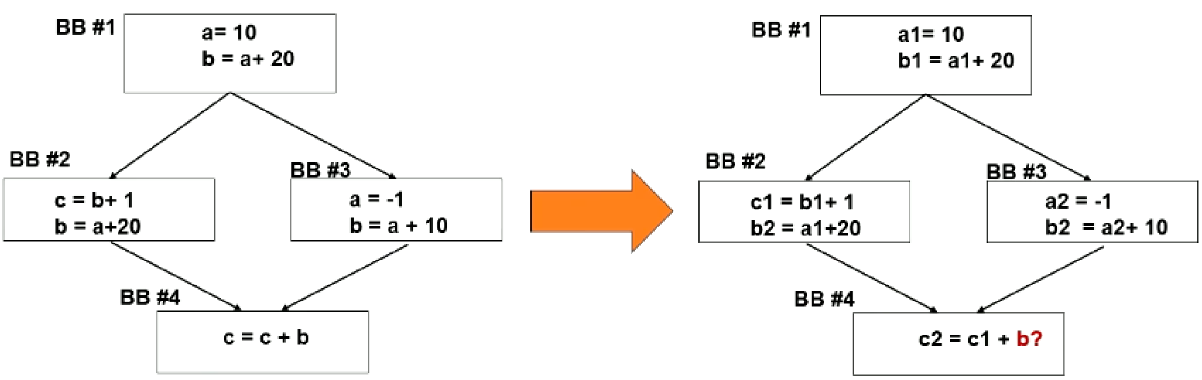
1. SSA is an enhancement of the def-use chain
2. Key feature: Variables can be defined only once in SSA form
3. Common usage: Intermediate Representations (IR) in compilers are typically in SSA form



SSA

SSA and Control Flow Graphs

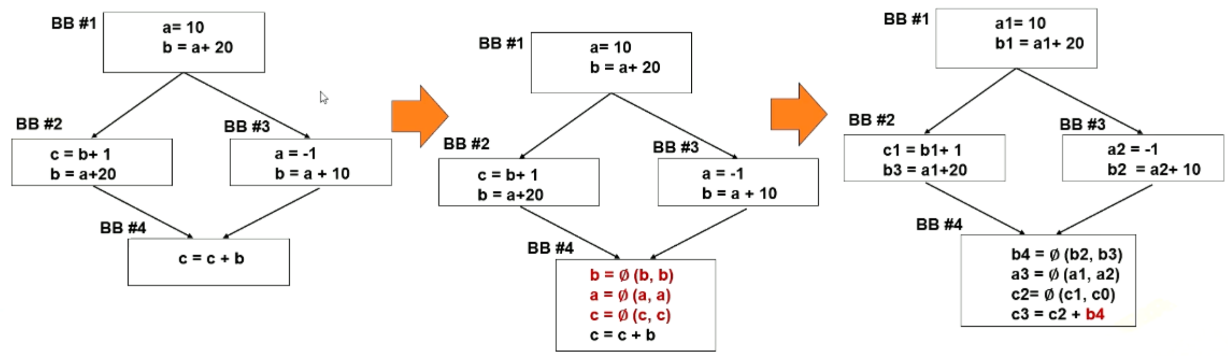
1. What if variable b is defined on both execution paths?



SSA

Phi - Function

1. Phi function merges values from different paths
2. Phi function can be implemented using move or other methods at the ISA level
3. Each definition gets a new version of the variable
4. Usage always uses the latest version
5. Phi function is added at each joint point for every variable
    - More than one predecessor



SSA

When to Insert Phi Function?

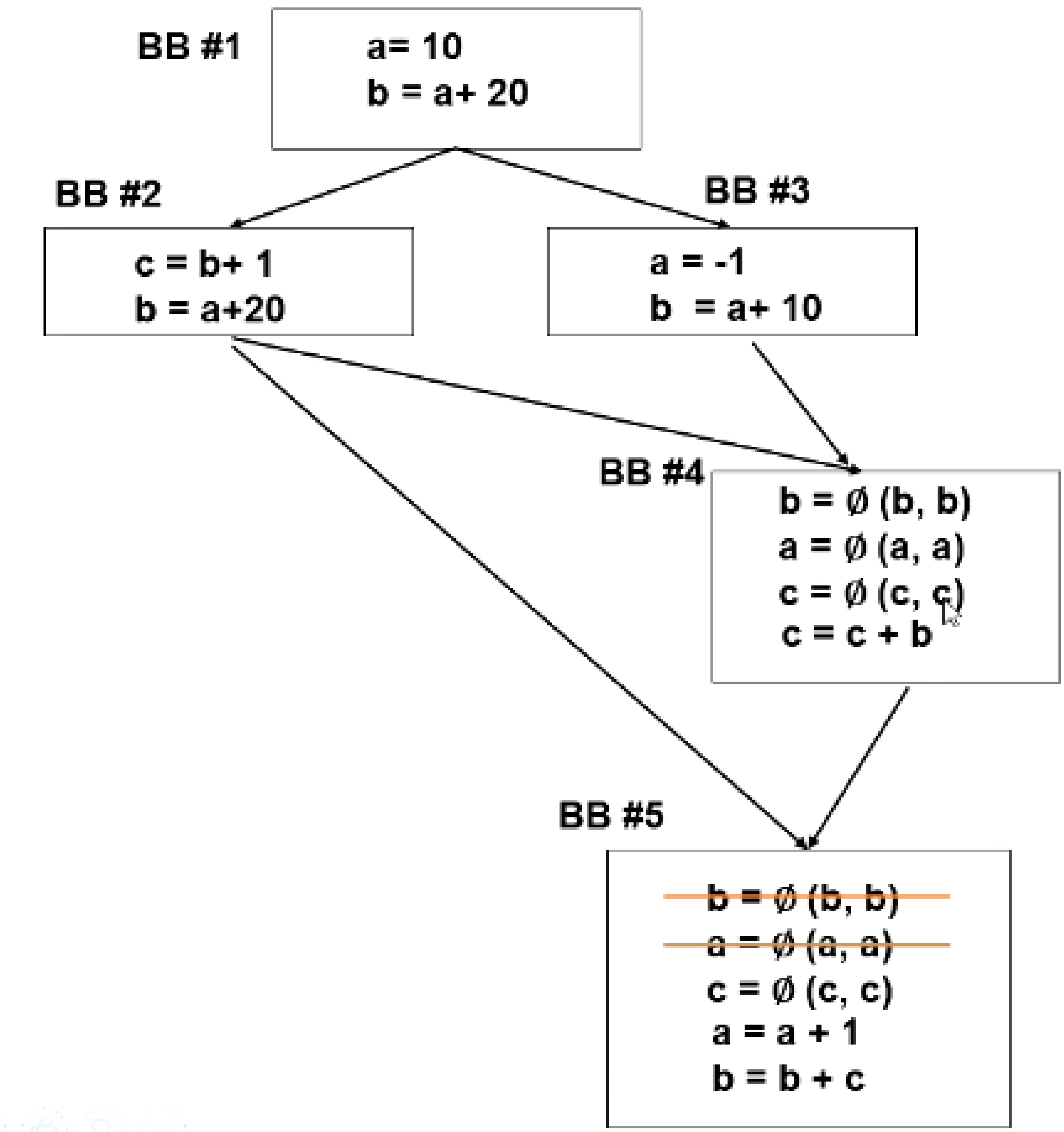
1. If Phi function is added at each joint point for every variable…
   - More than one predecessor -> can generate too many phi functions
2. Phi functions only need to be inserted when multiple values exist
3. Iterative path-convergence criterion needs to be considered

Path-Convergence Criterion

1. Phi function needs to be inserted when all the following are true
   - There is a block x containing a definition of a
   - There is a block y (y != x) containing a definition of a
   - There is a nonempty path Pxz of edges from x to z
   - There is a nonempty path Pyz of edges from y to z
   - Path Pxz and Pyz do not have any node in common other than z
   - The node z does not appear within both Pxz and Pyz prior to the end though it may appear in one or the other
2. Initialization: start node has all variable definitions

Applying Path-Convergence Criterion

1. From BB1 to BB2: path1,5 has common path BB4
2. Variable a and variable b are merged at BB4
3. But variable c is newly defined again



Applying Path-Convergence Criterion

Phi Function in LLVM

1. Actual phi function might be implemented with a branch instruction
   - Also could be a conditional move, depending on ISA

```c
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

```llvm
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:                                      ; preds = %2
  br label %end

bfalse:                                     ; preds = %2
  br label %end

end:                                        ; preds = %btrue, %bfalse
  %retval = phi i32 [%a, %btrue], [%b, %bfalse]
  ret i32 %retval
}
```

Phi Function in LLVM

Summary

1. SSA (static single-assignment) form enhances data-flow analysis by allowing variables to be defined only once
2. The Phi function is used to merge values from different paths in SSA form
3. The path-convergence criterion helps decide when to insert Phi functions during SSA conversion
4. SSA form simplifies data-flow analysis and enables more effective optimizations

Examples of Compiler Optimizations

Learning Objectives

1. Explain the fundamental concepts of compiler optimizations
2. Explore specific optimization techniques, including loop unrolling, function inlining, and more

Loop Unrolling

1. Unroll a loop for a small number of times for other code optimizations
2. Benefits:
    - Better instruction scheduling

- More opportunities for vectorization
- Reducing number of branches

Function Inlining

1. Interprocedure analysis
2. Benefits: Reduce the overhead of stack operations
3. Downside: Could increase code size

Dead Code Elimination

1. Removing code that has no effect on the program
2. The following code:

```
.entry main()
{
    .reg .f32 a, b, c;

    // Load values into registers
    ld.param.f32 a, [param_a];
    ld.param.f32 b, [param_b];

    // Multiply a and be, but the result is not used
    mul.f32 c, a, b;

    add.f32 c, a, 1.0;
    div.f32 c, b, 2.0;

    // Return
    ret;
}
```

3. Can be reduced to:

```
.entry main()
{
    .reg .f32 a, b, c;

    // Load values into registers
    ld.param.f32 a, [param_a];
    ld.param.f32 b, [param_b];

    add.f32 c, a, 1.0;
    div.f32 c, b, 2.0;

    // Return
    ret;
}
```

Constant Propagation

1. Technique that replaces variables with their constant values in the code, eliminating unnecessary variable accesses
2. Constant propagation often also triggers other optimizations

```
.entry main()
{
```

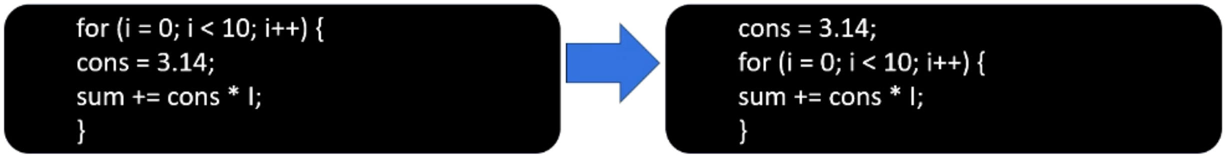
```
    .reg .f32 a, b, c;

    // Load values into registers
    mov.f32 a, 5.0;
    mov.f32 b, 3.0;

    // Multiply a and be, but the result is not used
    mul.f32 c, a, b;

    // Other instructions that use c
    add.f32 c, c, 1.0;

    // Store the result in memory
    st.global.f32 [result], c;

    // Return
    ret;
}

.entry main()
{
    .reg .f32 c;

    // Perform calculations with constants
    mul.f32 c, 5.0, 3.0;

    // Other instructions that use c
    add.f32 c, c, 1.0;

    // Store the result in memory
    st.global.f32 [result], c;

    // Return
    ret;
}
```

Strength Reduction

1. Replace expensive operations with cheaper operations
2. Divide by 2 -> right shift
3. Compute square operations -> multiplication

Code Motion: Loop-Invariant Instructions

1. Move code that is not dependent on loop operations
2. i variable is the loop induction variable
3. PTX: add.32 %r1, %r1, 1



Code Motion

Summary

1. Loop unrolling, function inlining, and strength reduction offer various compiler optimization opportunities
2. Reviewed examples of loop unrolling, function inlining, and strength reduction, dead code elimination, and constant propagation
3. Moving loop-invariant instructions is an example of code motion

Divergence Analysis

Learning Objectives

1. Describe static code analysis techniques to detect warp divergence

Review: Divergent Branches

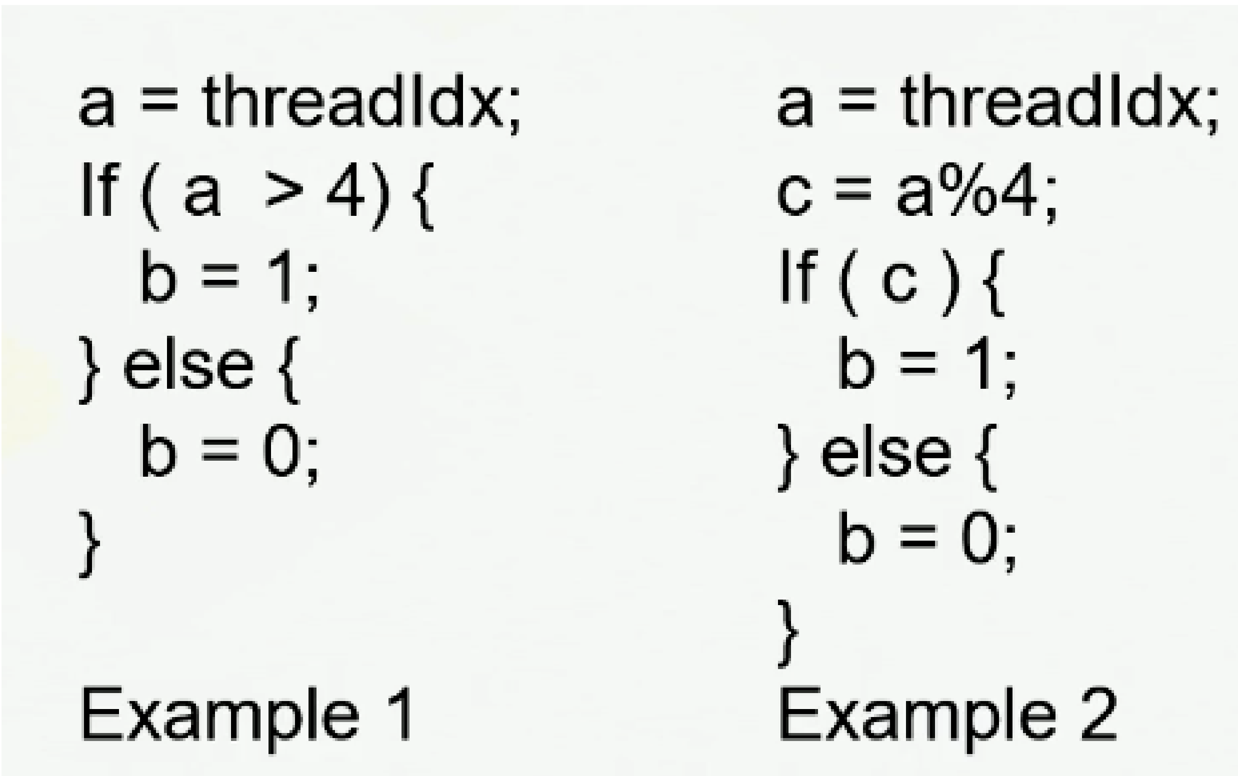
1. Within a warp, not all threads need to be executed
    - e.g., if-else statements
2. Not all threads will do the same work
3. Major difference between SIMD and SIMT
4. Compiler might need to insert reconvergence point, aka: IPDOM
    - IPDOM: Immediate post dominator

Divergence Analysis and Vectorization

1. Vectorization: converting loops with vector code (SIMD)
2. Vectorization requires checking whether all instructions will execute in lock-step
3. Warp divergence analysis shares similar characteristics: the need to check whether all threads within a warp will execute the same path
4. Complications: unstructed CFGs (complex control flow graphs)

Divergent Branches: Thread Dependency

1. An expression is thread dependent at a program point p if it may evaluate to different values by different threads at p
    - Example source code
    - If condition is dependent on thread
    - Main source of divergent branches



Divergent Branches

How to Check Thread Dependency?

1. Def-Use chain of thread IDs
2. Identify all dependences of thread IDs
3. Iterative search process similar to data flow analysis
4. Reachability of thread ID needs to be checked
5. Other example of divergence? What if the branch condition is coming from memory?
6. Challenges:
    - Complex control flow graphs
    - Too conservative analysis
        - All branches, all loops could be divergent

Summary

1. Reviewed conditions of warp-divergence check techniques
2. Branch that is dependent on thread ID is a divergent branch
3. An example of using data-flow analysis for GPU programming analysis