13th International Workshop on OpenCL and SYCL

IWOCL 2025

Efficient Barrier-related Debugging Information Reconstruction on the DPC++ CPU Device

Alexey Sachkov, Intel

Wenju He, Xinmin Tian, Senran Zhang, Intel.

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

- Agenda
  - Explain the problem.
  - Describe debug information construction algorithm in barrier fissioned work-group loop.
  - Describe optimization of debug information updating algorithm.
  - Show experiment results.
- Motivations
  - Efficient debugging of device code on the OpenCL CPU Device with good debugging performance.
- Two contributions (algorithms) in this paper
  - Reconstruct cross-barrier variables' debug information in LLVM IR by allocating a new alloca to store the cross-barrier value's address in the barrier *special buffer*.
  - Propose a novel *barrier region* based on LLVM basic block dominance frontiers, to minimize the number of instructions created to calculate a variable's address in the *special buffer* and store the address into the new alloca.
- Note
  - The algorithms are specifically designed for the case of barrier work-group loop fission on CPU device of OpenCL, SYCL, etc.

Background: Work-group Loop on OpenCL CPU Device

- On the Intel OpenCL CPU Device, a work-group is mapped into a CPU thread.
- A loop is constructed over work-items in a work-group.
- Work-items in a work-group are executed in ascending order of local work-item id.

```
#include <sycl/sycl.hpp>

int main() {
  using namespace sycl;
  size_t N = 1024;
  queue Q;
  auto *A = malloc_shared<int>(N, Q);
  try {
    Q.submit([&](handler &CGH) {
      CGH.parallel_for<class kernel>(nd_range<1>(N, 32), [=](nd_item<1> It) {
        A[It.get_global_id(0)] = 0;
      });
    }).wait();
  } catch (sycl::exception const &) {
  }
  free(A, Q);
  return 0;
}
```

DPC++ CPU Compiler

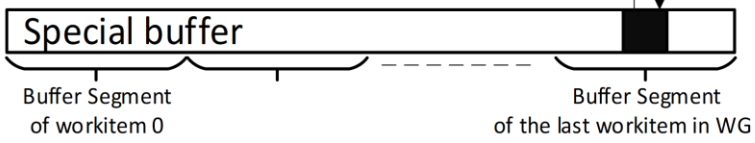
```
Loop over a work-group


void _ZTSN4sycl3_V16kernelE(...) {
  id = get_group_id(0) * get_local_range(0);
  for (lid = id; lid < id + get_local_range(0); ++lid)
    A[lid] = 0;
}
```

Barrier Work-group Loop Fission

See Evgeniy Tyurin's talk from 2018 Bay Area LLVM Dev's conference about barrier implementation in OpenCL CPU Device: video slides

```
CGH.parallel_for<class kernel>(sycl::nd_range<1>(N, 32), [=](sycl::nd_item<1> It) {
   auto GlobalId = It.get_global_id(0);
   int V = A[GlobalId];
   It.barrier(sycl::access::fence_space::global_space);
   B[GlobalId] = V;
});
```

```
int LocalId = 0;
int CurrBarrierId = 0;
label0:
 int V = A[GlobalId];
 store V into special buffer[offset]
 if(LocalId < WorkGroupSize) {
   LocalId++;
   switch (CurrBarrierId) {
     case 0: goto label0;
     default: goto label1;
   }
 } else {
   LocalId = 0;
   CurrBarrierId = 1;
 }
label1:
 load V from special buffer[offset]
 B[GlobalId] = V;
```



- Loop fission causes problem for maintaining debug information:
  - The debug information for values moved into the special buffer is lost because the original instruction holding the information becomes obsolete.
  - Since the special buffer is created as a whole using a single LLVM alloca instruction, it cannot retain debug information for all the variables it holds.

```
6  auto *A = sycl::malloc_shared<int>(N, Q);
7  for (size_t I = 0; I < N; ++I)
8    A[I] = (int)I + 16;

12    CGH.parallel_for<class kernel>(sycl::nd_range<1>(N, 32), [=](sycl::nd_item<1> It) {
13       auto GlobalId = It.get_global_id(0);
14       int V = A[GlobalId];
15       It.barrier(sycl::access::fence_space::global_space);
16       B[GlobalId] = V;
17    });
```

Before debug info reconstruction
Bad

Before Barrier

```
(gdb) b 14
Breakpoint 1 at 0x403122: file test.cpp, line 14.
(gdb) r
Thread 28 "a.out" hit Breakpoint 1, main::{...It=...) at barrier.usm.cpp:14
14              int V = A[GlobalId];
(gdb) p GlobalId
$1 = 0
(gdb) n
15              It.barrier(sycl::access::fence_space::global_space);
(gdb) p V
$2 = <optimized out>
(gdb) n
13              auto GlobalId = It.get_global_id(0);
(gdb) n
Thread 28 "a.out" hit Breakpoint 1, main::{...It=...) at barrier.usm.cpp:14
14              int V = A[GlobalId];
(gdb) p GlobalId
$3 = 1
(gdb) n
15              It.barrier(sycl::access::fence_space::global_space);
(gdb) p V
$4 = <optimized out>
```

work-item 0

work-item 1

After Barrier

```
(gdb) d 1
(gdb) b 16
Breakpoint 2 at 0x403141: /localdisk2/wenjuhe/tmp1/barrier.usm.cpp:16.
(gdb) c
Continuing.

Thread 28 "a.out" hit Breakpoint 2, main::{...It=...) at barrier.usm.cpp:16
16              B[GlobalId] = V;
(gdb) p GlobalId
$5 = 63
(gdb) p V
$6 = <optimized out>
(gdb) n
17            });
(gdb) n

Thread 28 "a.out" hit Breakpoint 2, main ::{...It=...) at barrier.usm.cpp:16
16              B[GlobalId] = V;
(gdb) p GlobalId
$7 = 63
(gdb) p V
$8 = <optimized out>
```

Garbage value

Debug Info Reconstruction in Barrier

- Reconstruction algorithm
  - Create a new alloca instruction for each alloca and function argument that has debug information.
  - A new llvm.dbg.declare instruction is created for the new alloca and inserted in entry basic block. The DW_OP_deref operation is prepended to the DIExpression of the llvm.dbg.declare instruction to dereference the pointer and obtain the original value.
  - At the use sites, we store the value's address in the special buffer into the alloca and attach debug information to the new alloca.
  - The user of cross-barrier value is replaced with the address in the special buffer.

```
auto GlobalId = It.get_global_id(0);
int V = A[GlobalId];
It.barrier(fence_space::global_space);
B[GlobalId] = V;
```

LLVM IR before barrier handling

```
entry:
  %GlobalId = alloca i64, align 8
    #dbg_declare(ptr %GlobalId, !399, !DIExpression(DW_OP_constu, 4,
                 DW_OP_swap, DW_OP_xderef), !409)
  %V = alloca i32, align 4
    #dbg_declare(ptr %V, !39, !DIExpression(DW_OP_constu, 4,
                 DW_OP_swap, DW_OP_xderef), !23)
  %0 = load ptr addrspace(4), ptr addrspace(4) %A, align 8, !dbg !24
  %1 = load i64, ptr %GlobalId, align 8, !dbg !25
  %arrayidx = getelementptr inbounds i32, ptr addrspace(4) %0, i64 %1
  %2 = load i32, ptr addrspace(4) %arrayidx, align 4, !dbg !24
  store i32 %2, ptr %V, align 4, !dbg !23
  br label %Split.Barrier.BB, !dbg !26
Split.Barrier.BB:                                 ; preds = %entry
  call void @_Z18work_group_barrierj(i32 1), !dbg !26
```

LLVM IR after barrier handling

```
entry:
  %V.addr = alloca ptr, align 8
  %GlobalId.addr = alloca ptr, align 8
SyncBB2:
  %SB_Offset = add nuw i64 %SB_Index, 48, !dbg !31
  %SB_V = getelementptr inbounds i8, ptr %SB, i64 %SB_Offset, !dbg !31
  store ptr %SB_V, ptr %V.addr, align 8, !dbg !31
    #dbg_declare(ptr %V.addr, !43, !DIExpression(DW_OP_deref, DW_OP_constu, 0,
                                                 DW_OP_swap, DW_OP_xderef), !44)
  %SB_Offset2 = add nuw i64 %SBIndex, 56, !dbg !41
  %SB_GlobalId = getelementptr inbounds i8, ptr %SB, i64 %SB_Offset2, !dbg
  store ptr %SB_GlobalId, ptr %GlobalId.addr, align 8, !dbg !41
    #dbg_declare(ptr %GlobalId.addr, ……
  %4 = load ptr addrspace(4), ptr addrspace(4) %A, align 8, !dbg !33
  %5 = load ptr, ptr %SB_GlobalId, align 8, !dbg !14
  %arrayidx = getelementptr inbounds i32, ptr addrspace(1) %4, i64 %5, !dbg !24
  %6 = load i32, ptr addrspace(1) %arrayidx, align 4, !dbg !46
  store i32 %7, ptr %SB_V, align 4, !dbg !28
```

```
6  auto *A = sycl::malloc_shared<int>(N, Q);
7  for (size_t I = 0; I < N; ++I)
8    A[I] = (int)I + 16;

12    CGH.parallel_for<class kernel>(sycl::nd_range<1>(N, 32), [=](sycl::nd_item<1> It) {
13       auto GlobalId = It.get_global_id(0);
14       int V = A[GlobalId];
15       It.barrier(sycl::access::fence_space::global_space);
16       B[GlobalId] = V;
17    });
```

After debug info reconstruction
Good

Before Barrier

```
(gdb) b 14
Breakpoint 1 at 0x403122: file test.cpp, line 14.
(gdb) r
Thread 28 "a.out" hit Breakpoint 1, main::{...It=...) at barrier.usm.cpp:14
14              int V = A[GlobalId];
(gdb) p GlobalId
$1 = 0
(gdb) n
15              It.barrier(sycl::access::fence_space::global_space);
(gdb) p V
$2 = 16
(gdb) n
13              auto GlobalId = It.get_global_id(0);
(gdb) n
Thread 28 "a.out" hit Breakpoint 1, main::{...It=...) at barrier.usm.cpp:14
14              int V = A[GlobalId];
(gdb) p GlobalId
$3 = 1
(gdb) n
15              It.barrier(sycl::access::fence_space::global_space);
(gdb) p V
$4 = 17
```

work-item 0

work-item 1

After Barrier

```
(gdb) d 1
(gdb) b 16
Breakpoint 2 at 0x403141: /localdisk2/wenjuhe/tmp1/barrier.usm.cpp:16.
(gdb) c
Continuing.

Thread 28 "a.out" hit Breakpoint 2, main::{...It=...) at barrier.usm.cpp:16
16              B[GlobalId] = V;
(gdb) p GlobalId
$5 = 0
(gdb) p V
$6 = 16
(gdb) n
17            });
(gdb) n

Thread 28 "a.out" hit Breakpoint 2, main ::{...It=...) at barrier.usm.cpp:16
16              B[GlobalId] = V;
(gdb) p GlobalId
$7 = 1
(gdb) p V
$8 = 17
```

Optimization of Debug Info Reconstruction in Barrier

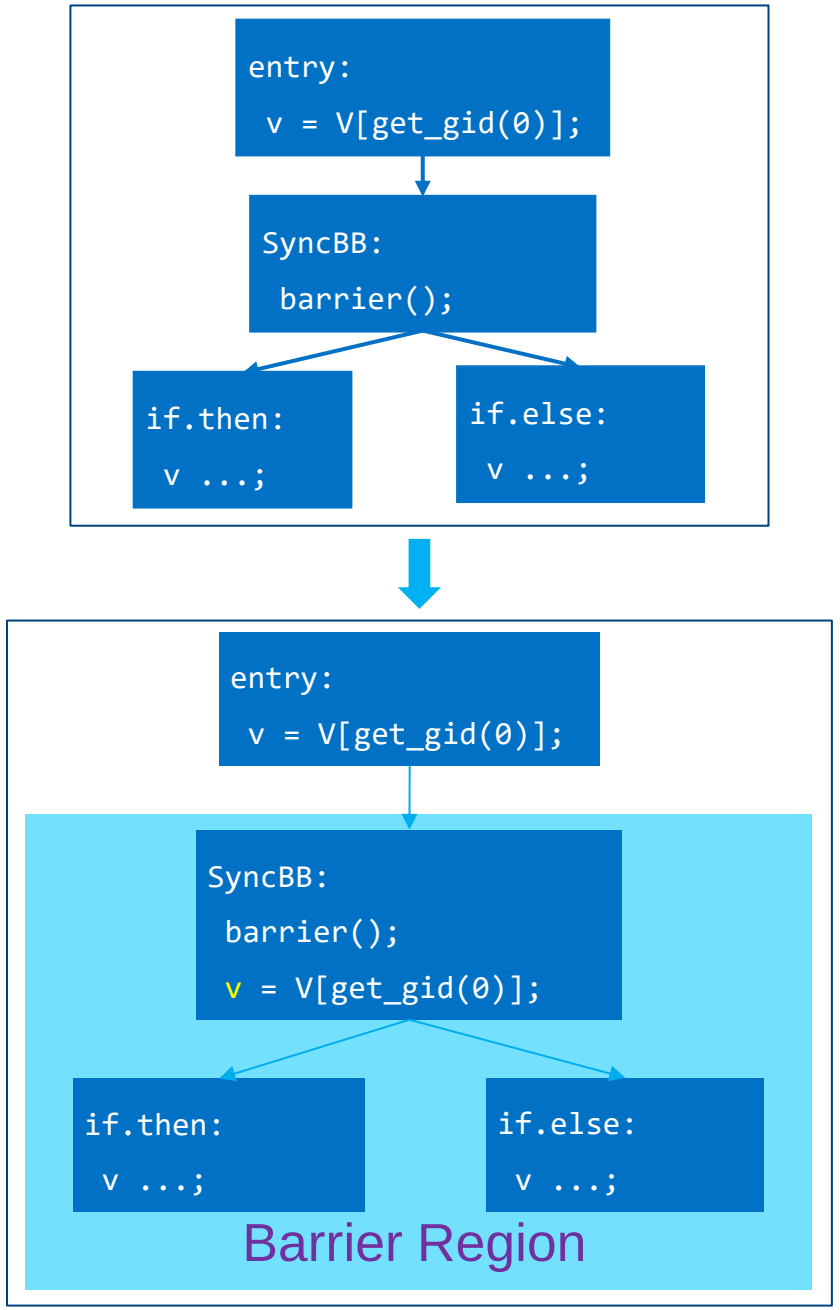
- To minimize the number of instructions to compute value's address and store value address to the newly created allocations, we propose a novel approach that analyses barrier region based on the LLVM basic block dominance frontier.

- Region may have different meanings. We clarify barrier region is not the set of ***all*** basic blocks in a fissioned work-group loop. In a complex CFG, a fissioned work-group loop may consist of multiple barrier regions.

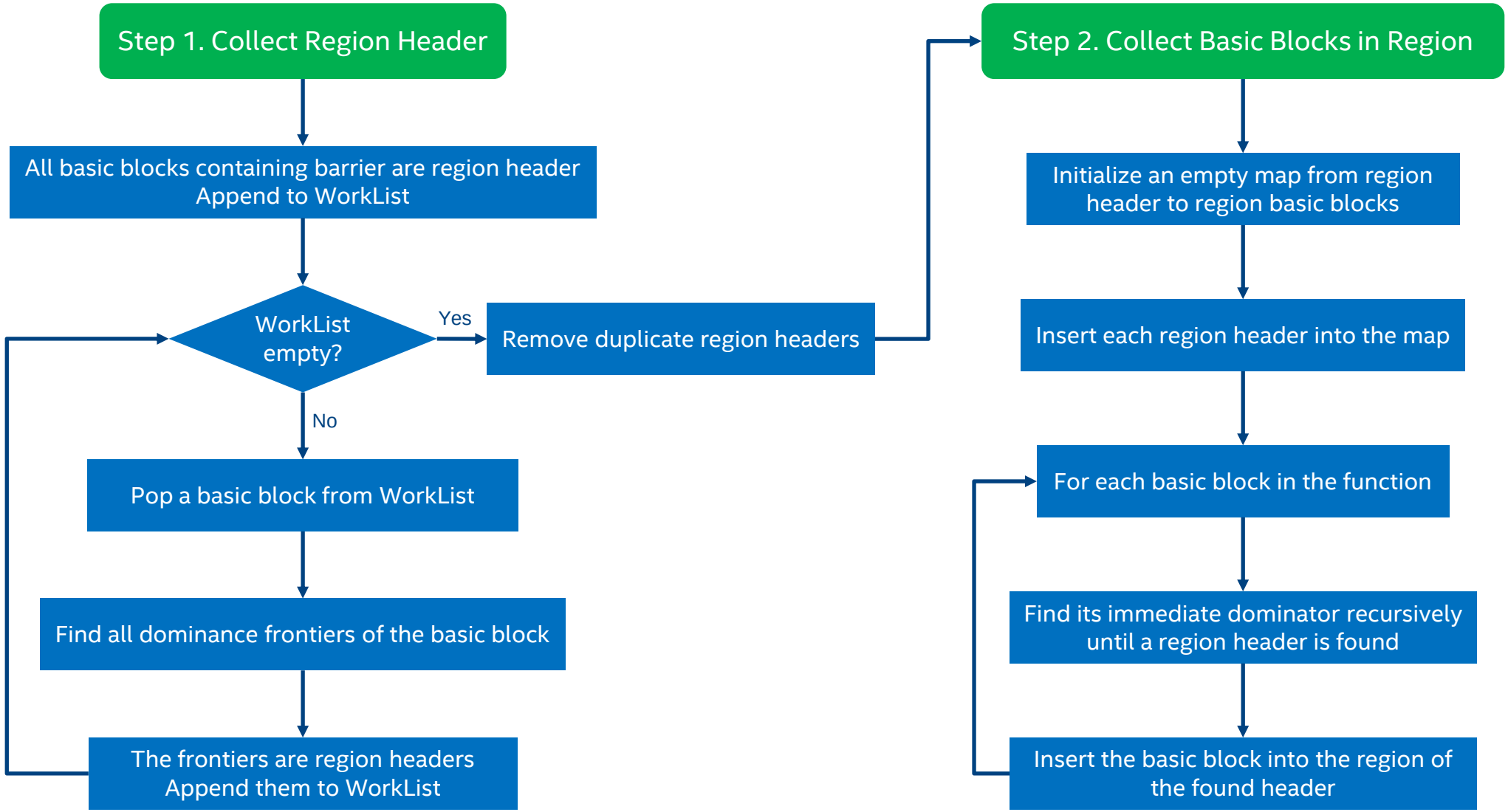
- barrier region is a very generic optimization for CPU Device. It accurately models a group of basic blocks that there won't be a barrier between two values defined in the region. A cross-barrier value's address in special buffer doesn't change within a barrier region.
  - It can also be used for other optimizations. For example, another application of barrier region is eliminating cross-barrier value by copying value definition into *barrier region* header, results in up-to 40% performance gain on OpenCL specACC/125.
  - It can be adopted for other OpenCL or SYCL CPU devices.

Barrier Region

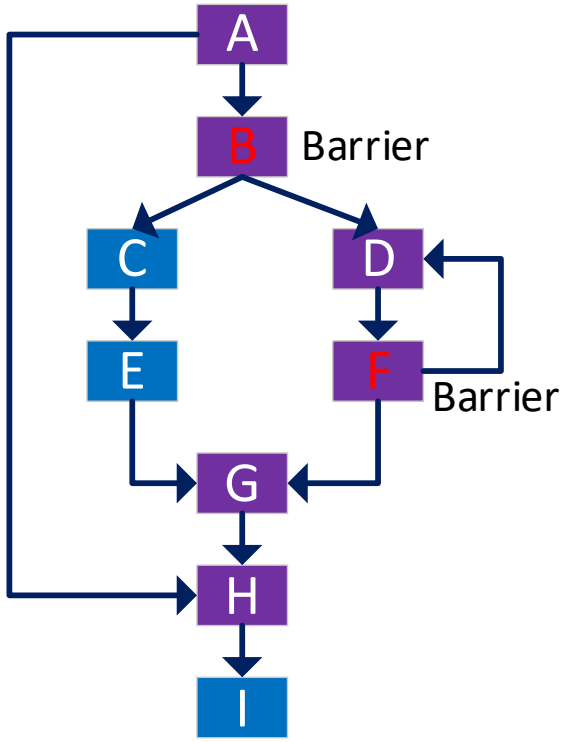
- A cross-barrier value shares a same definition inside a region.
- Barrier region
  - A barrier region starts with a region header and contains a set of basic blocks.
  - A non-header basic block belongs to a barrier region iff its immediate dominator
    - is the region header, or
    - belongs to the region.
- Barrier region header
  - A basic block is a barrier region header iff it's
    - the entry block of a function, or
    - a sync BB (Basic block's first instruction is barrier), or
    - a dominance frontier of another region header.



Barrier Region Construction Algorithm



Barrier Region Example



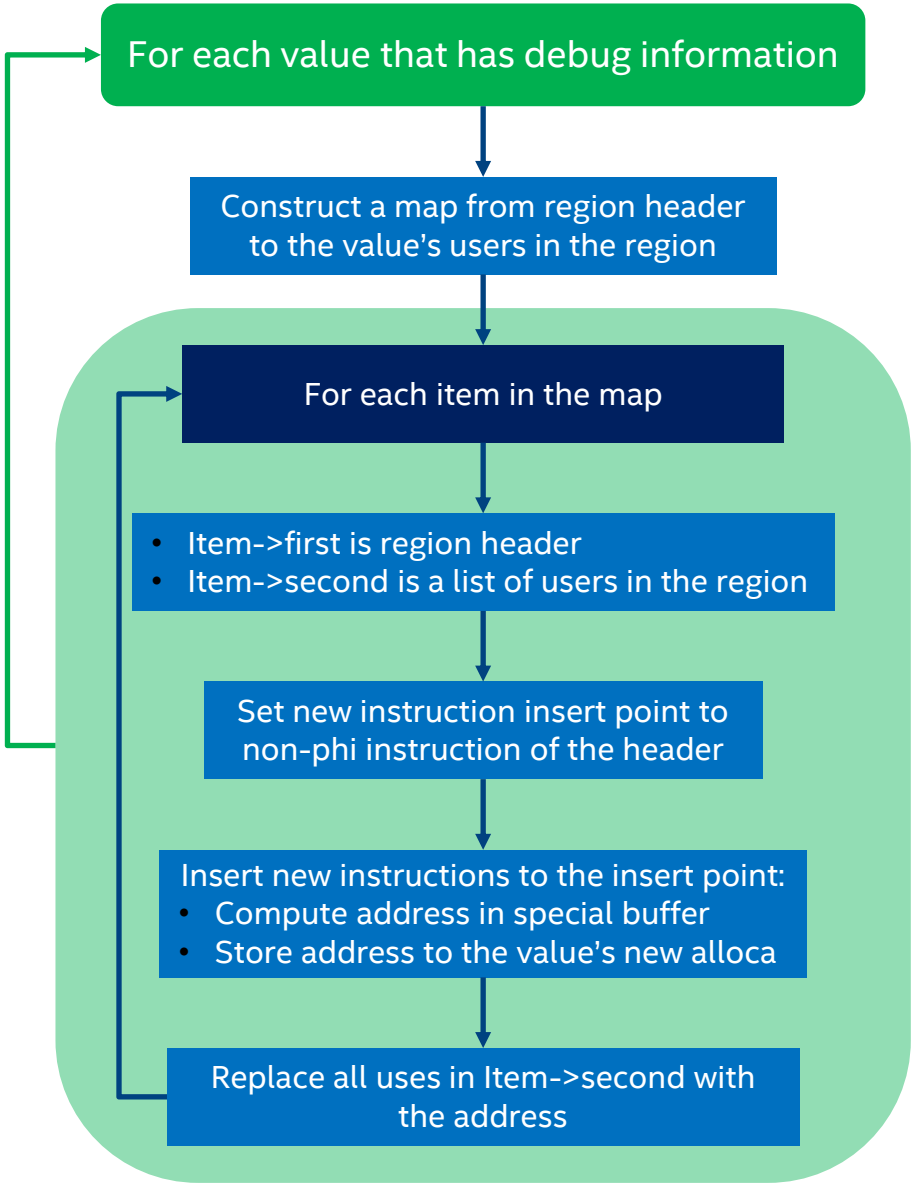
Example of Control Flow Graph

- Basic block **B** and **F** are sync BB (the first inst is barrier)
- Region headers (in purple color)
- Region basic blocks (in blue color)

Block	A	B	C	D	E	F	G	H	I
Dominance frontiers	ø	H	ø	D,G	ø	D,G	H	ø	ø

Region header	A	B	D	F	G	H
Why is header?	Entry BB	SyncBB	DF of F	SyncBB	DF of F	DF of G
BBs in Region	A	B,C,E	D	F	G	H, I

Update Debug Information using Barrier Region



Experiment

- Compile option: -g -O0
- Functionally, we validated our approach on our internal SYCL debugging test suites, and the debug information passed verification.
- Reduction rate in the number of LLVM IR instructions
  - 0.2% to 1.8% in six SYCL benchmarks that have barrier: Apriori, Kmeans, Md3p, Rodinia, SGEMM, SVM, Md3p.
  - 24% in an OpenCL benchmark SPEC ACCEL/124, i.e., “Structured Grid, Physics Simulation”.
- Performance gain
  - System configuration
    - Operating System: Red Hat Enterprise Linux release 9.0
    - Processor: Intel ICX 8358 128-core CPU
  - 1.9% gain on SYCL benchmark SGEMM
    - Number of runs: 9. The gain is stable.
  - 19.1% gain on OpenCL benchmark SPEC ACCEL/124
    - Number of runs: 3. The gain is stable.

Summary

- We presented an algorithm for reconstructing debug information for cross-barrier value in the case of barrier loop fission.
- We presented a novel algorithm for minimizing number of insert points for debug information reconstruction.
- These two algorithms are implemented in the latest Intel oneAPI toolkit.
- Key benefits:
  - Improved debugging experience
  - Smaller code size and better performance
- Can be adopted to other CPU devices of OpenCL, SYCL, etc.

Thank you

Backup Slides

Further Enhancement

- To refine following case, use LLVM RegionInfo to gather single-entry single-exit canonical regions within a barrier region.

```
for.cond:
  br i1 %cmp, label %for.body, label %for.end
for.body:
  br %master.thread.fallthru
master.thread.fallthru:
  call void @_Z18work_group_barrierj(i32 1)
  br label %for.inc
for.inc:
  br label %for.cond
for.end:
  store i32 0, ptr %j, align 4
```

- In rare case, where there is an unreachable basic block, the algorithm may fail. Then, we take fallback approach that stores the special buffer address into the new alloca at every use site.