



Scalarization across threads

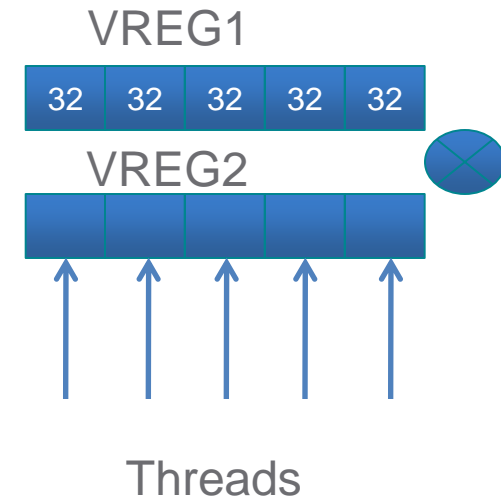
Alexander Timofeev
MARCH 2016

Goals and assumptions

- Architecture with massive data parallelism
- Has both scalar and vector units
- Not all the data flow is naturally vector
- Goal is to split vector and scalar flow and replace vector operations with scalar operations where it is possible
- We save VREGs to those operations which really need them and speed up their execution with more threads.

Abstract parallel machine: registers

- Vector unit
 - operates on vector registers
 - N 32bit lanes – one lane per thread
 - executes vector instructions
 - executes N threads in parallel
- Scalar Unit
 - operates on 32bit scalar registers
 - executes scalar instructions
- Scalar to vector value broadcast is cheap
- Divergent Control Flow is expensive



- Private memory
 - each of N threads has dedicated private memory area
 - other threads have no access to the thread private memory
- Shared memory
 - shared among the N threads
 - 2 threads executing memory operation on shared memory
 - considered to access same value if the effective addresses in both memory operations are the same
 - scalar and vector data caches are not necessarily coherent: writing value to shared memory via vector instruction does not invalidate respective scalar cache line, and vice versa

Scalarization across threads

- What if at some point all the lanes of vector instruction operand contain equal values?
- In this case we say that operand is Uniform
- Vector operation taking uniform operands produces uniform result
- We could change uniform vector operation to scalar
 - Saving VGPRs – more threads in parallel
 - Scalar L1 cache latency is smaller
 - Saturating SALU – more opportunities for the scheduler
- The goal of the analysis is to split scalar and vector data/control flows

Scalarization: read-only property

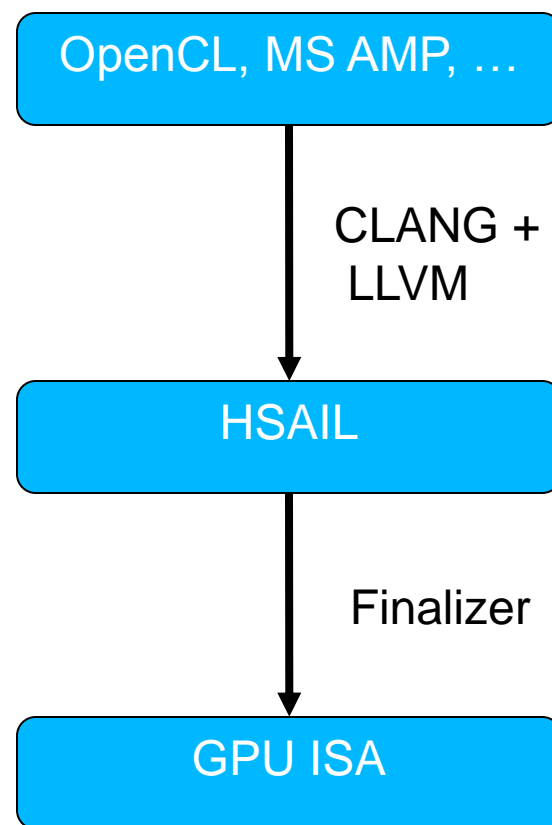
- Scalar and vector caches are not coherent
 - Write to an address via vector unit does not invalidate scalar cache line corresponding to this address
 - Changing vector memory operation to scalar is safe if and only if we prove that corresponding memory location cannot be written by another instruction
- Scalarization requires read-only property
 - Read-only memory: “__constant” or “__readonly” in OpenCL
 - Read-only modifier: “const” for arguments
 - Proven no writes in concrete memory location over all paths from the function entry to the read point: trace analysis + AA

Scalarization: structure

- Operation uniformity analysis
 - Performed by LLVM-based high-level compiler
 - Implemented as a custom module pass
 - Result: each operation is attributed by the special value defining its 'width'
- Constancy analysis: each operation is attributed with logical value defining if it may be written
- Transformation itself is performed according to the 'width' and 'const' attributes of the operation.

Current implementation restrictions: AMD specific

- AMD HSA compiler consists of 2 levels:
 - LLVM-based high-level compiler generates HSAIL
 - Low-level Finalizer accepts HSAIL and generates GPU ISA
- HSAIL by design does not assume vector flow: all the abstract registers are scalar
- 'Width' and 'Const' attributes are passed through the HSAIL to Finalizer
- Finalizer performs scalarization according to the passed attributes



Current implementation restrictions: LLVM specific



- We collect information over IR
- We apply information on Machine Code
- LLVM has no support for passing additional information over ISel: metadata is insufficient
- In upcoming AMDGPU compiler we would explicitly select vector or scalar form for instruction according the collected 'width' and 'const' attributes

Data dependency

```
__global int * A;  
int x, y;  
x=get_global_id(0);  
y=A[x];
```

- Explicitly reflected in SSA form
- Thread-specific data introduced by restricted set of operations

Control dependency

```
__global int * A;  
int idx, n=...(input);  
if(n < get_global_id(0))  
    idx = 10;  
else  
    idx = 20;  
return A[idx];
```

- Is not reflected explicitly
- Needs some bookkeeping

- All width values are ordered and form trivial semi-lattice
 - 'thread' – vector operation – lattice bottom element
 - 'group' – operation is uniform for N-wide group
 - 'all' – scalar – lattice top element
- Let $W(x)$ be width function of operation x such as:

$W(x)$ is defined upon the poset W such as

$$W: \{\perp < w_0 < w_1 < \dots < \top\}$$

$$W(x) = \bigwedge_{y \in O(x)} W(y)$$

where $O(x)$ is x 's operands set

and $\bigwedge\{A\}$ is MIN over elements of A

$$\forall x \in W \quad \perp \wedge x = \perp, \quad \top \wedge x = x$$

$$W(\text{constant value}) = \top, \quad W(\text{kernel argument}) = \top$$

$$W(x) = \perp \quad \forall x \in E \text{ where } E \text{ is a set of non-uniform operations}$$

Data dependency analysis

- At the analysis start :
 - $W(x) = \top \quad \forall x \notin E$ and $W(x) = \perp \quad \forall x \in E$
- Operation 'width' is MIN over all operands 'width'
- MIN is monotonic, set is ordered and restricted
- Iterative analysis is proven to reach fixed point

Control dependency analysis

- Basic block post-dominance frontier forms a set of blocks of which the given one is control-dependent
- Post-dominance frontiers are computed by fast Cooper's algorithm

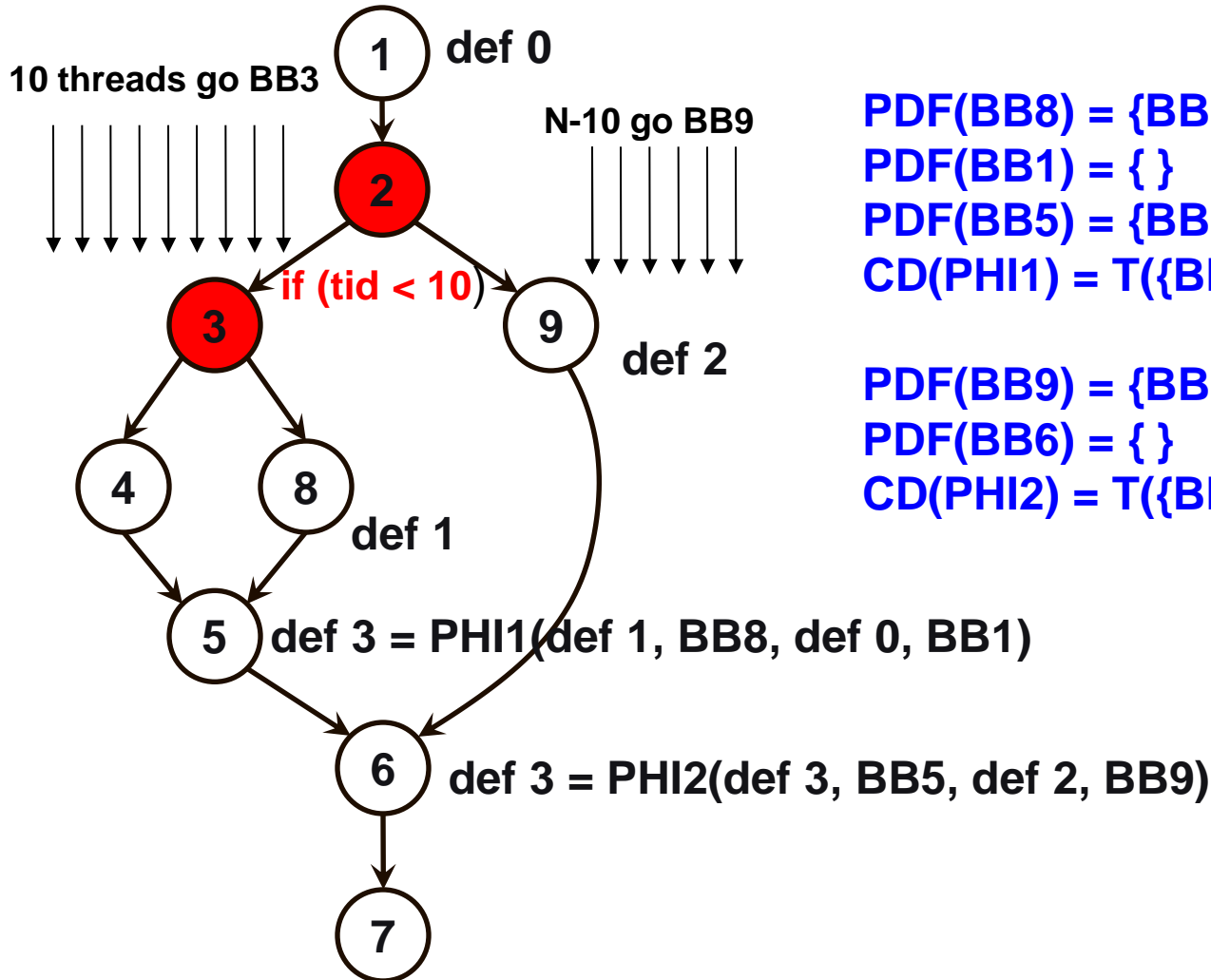
```
for (auto & B : F->getBasicBlockList())
{
    const TerminatorInst * T = B.getTerminator();
    if (T->getNumSuccessors() > 1)
    {
        succ_iterator I = succ_begin(&B);
        succ_iterator E = succ_end(&B);
        for ( ; I!=E; ++I)
        {
            DomTreeNode * runner = PDT->getNode(*I);
            DomTreeNode * sentinel = PDT->getNode(&B)->getIDom();
            while (runner && runner != sentinel)
            {
                functionsPDF[F][runner->getBlock()].insert(&B);
                runner = runner->getIDom();
            }
        }
    }
}
```

Control dependency analysis

- SSA-form makes value merge points explicit: PHI-nodes
 - *Let I – set of all function body instructions*
 - *Let $PHI \subset I$ – set of the all φ – functions*
 - *Let BB – set of all Basic Blocks of the Function*
 - *Let $B(i) : i \in I \rightarrow b \in BB$ gets parent block for instruction*
 - *Let $O(i) : i \in I \rightarrow \{j \mid j \in I \text{ and } j \text{ defines operand of } i\}$*
 - *Let T – set of all terminators and $T(x) : b \in BB \rightarrow \{t \mid t \in T\}$*
 - *Let $CD(\varphi) : \varphi \in PHI \rightarrow \{i \mid i \in I\}$*
 - *Let $PDF(bb) : bb \in BB \rightarrow \{b \mid b \in BB\}$*

- During iterative analysis for each PHI-node:
 - Compute the set of conditional branches for all φ operands as follows:
 - $CD(\varphi) = \bigcup_{o \in O(\varphi)} T[PDF(B(o)) \setminus PDF(B(\varphi))]$
 - Add them as pseudo-operands to the PHI $O(\varphi) = O(\varphi) \cup CD(\varphi)$
 - **Compute resulting φ – node ‘width’ as for usual operation**

Control dependency analysis



$$\text{PDF}(\text{BB8}) = \{\text{BB3}, \text{BB2}\}$$

$$\text{PDF}(\text{BB1}) = \{\}$$

$$\text{PDF}(\text{BB5}) = \{\text{BB2}\}$$

$$\text{CD}(\text{PHI1}) = \text{T}(\{\text{BB3}, \text{BB2}\} \setminus \{\text{BB2}\}) = \text{T}(\text{BB3})$$

$$\text{PDF}(\text{BB9}) = \{\text{BB2}\}$$

$$\text{PDF}(\text{BB6}) = \{\}$$

$$\text{CD}(\text{PHI2}) = \text{T}(\{\text{BB2}\})$$

Putting things together

- Walk call graph in post-order:
 - Callee is processed before caller
 - Each by-reference argument is attributed with 'width' to track non-uniform changing of pointers passed in
 - Call site analysis may lead to callee re-computation if we pass non-uniform value as an actual argument
- For each node in a CG iterative analysis produces attributed IR
- Further scalarization is performed according to the attributes

Example

OpenCL code

```
__kernel void test(__global int * in1, __constant int * in2, __global int * out, int n)
{
    int tid = get_global_id(0);
    for (int i=0; i<n; i++)
    {
        out[tid] = in1[tid%n] + in2[i] / in2[n%i];
    }
}
```

Example

Control flow graph

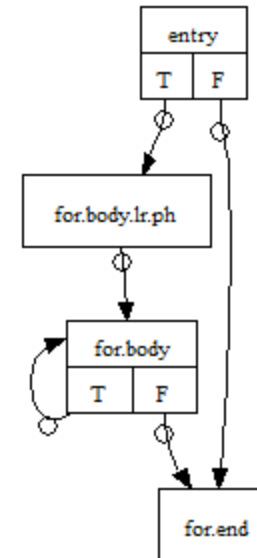
```
entry:
%0 = tail call spir_func i64 @__hsail_id_kernarg_u64(i32 0) #1
%1 = tail call spir_func i32 @__hsail_get_global_id(i32 0) #1
%2 = zext i32 %1 to i64
%3 = add i64 %2, %0
%conv = trunc i64 %3 to i32
%cmp1 = icmp sgt i32 %n, 0
br i1 %cmp1, label %for.body.lr.ph, label %for.end
```

```
for.body.lr.ph:
%rem = srem i32 %conv, %n
%idxprom = sext i32 %rem to i64
%arrayidx = getelementptr inbounds i32 @drrspace(1)* %in1, i64 %idxprom
%idxprom7 = sext i32 %conv to i64
%arrayidx8 = getelementptr inbounds i32 @drrspace(1)* %out, i64 %idxprom7
br label %for.body
```

```
for.body:
%lsriv1 = phi i32 @drrspace(2)* [ %scevgep, %for.body ], [ %in2, ... %for.body.lr.ph ]
%lsriv = phi i32 [ %lsriv.next, %for.body ], [ %n, %for.body.lr.ph ]
%i.02 = phi i32 [ 0, %for.body.lr.ph ], [ %inc, %for.body ]
%4 = load i32 @drrspace(1)* %arrayidx, align 4, !tbaa !9
%5 = load i32 @drrspace(2)* %lsriv1, align 4, !tbaa !9
%rem4 = srem i32 %n, %i.02
%idxprom5 = sext i32 %rem4 to i64
%arrayidx6 = getelementptr inbounds i32 @drrspace(2)* %in2, i64 %idxprom5
%6 = load i32 @drrspace(2)* %arrayidx6, align 4, !tbaa !9
%div = sdiv i32 %5, %6
%add = add nsw i32 %div, %4
store i32 %add, i32 @drrspace(1)* %arrayidx8, align 4, !tbaa !9
%inc = add nsw nsw i32 %i.02, 1
%lsriv.next = add i32 %lsriv, -1
%scevgep = getelementptr i32 @drrspace(2)* %lsriv1, i64 1
%exitcond = icmp eq i32 %lsriv.next, 0
%negate_loop_exit_cond = xor i1 %exitcond, true
br i1 %negate_loop_exit_cond, label %for.body, label %for.end
```

```
for.end:
ret void
```

CFG for '__OpenCL_test_kernel' function

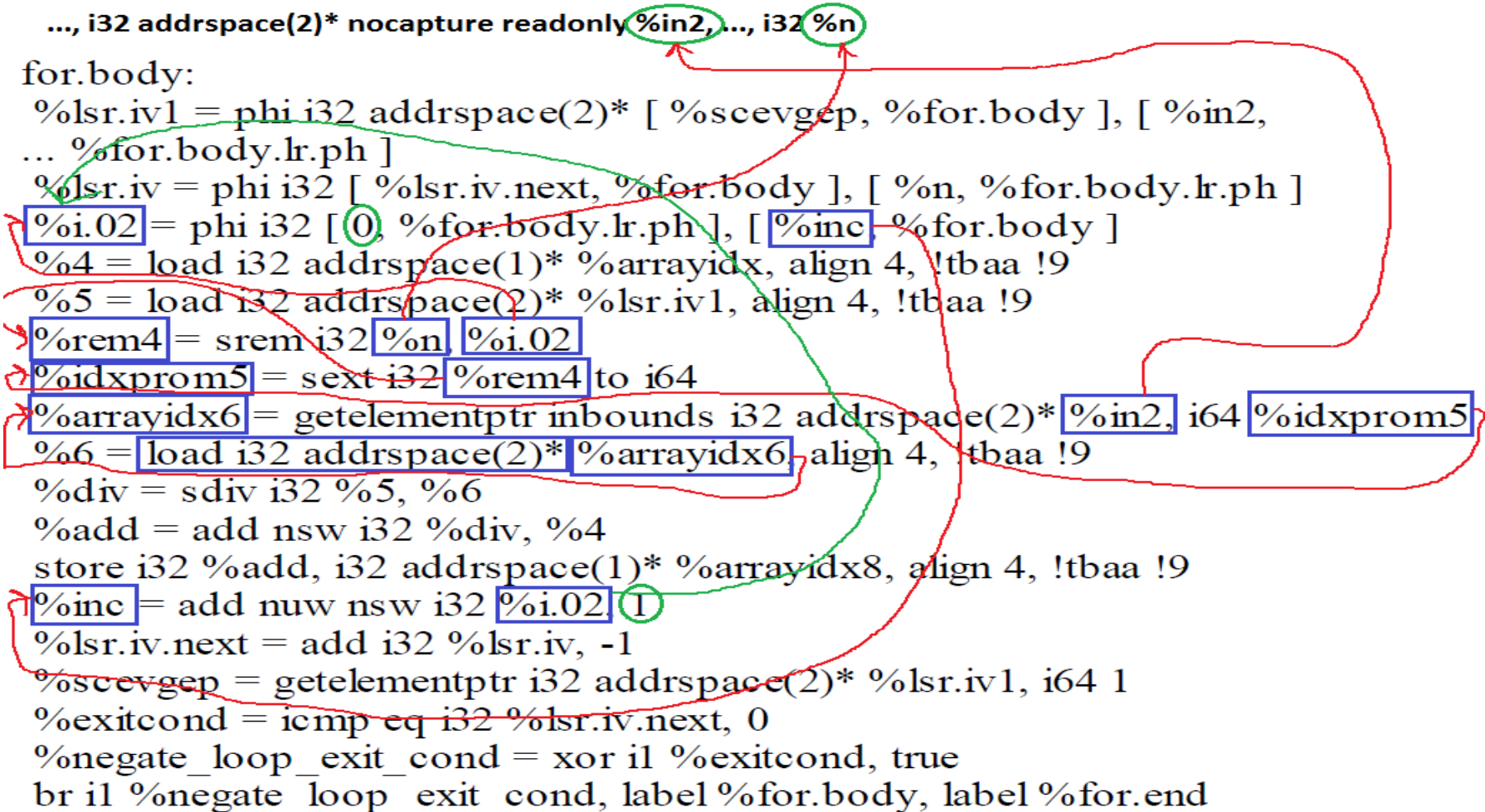


CFG for '__OpenCL_test_kernel' function

Example

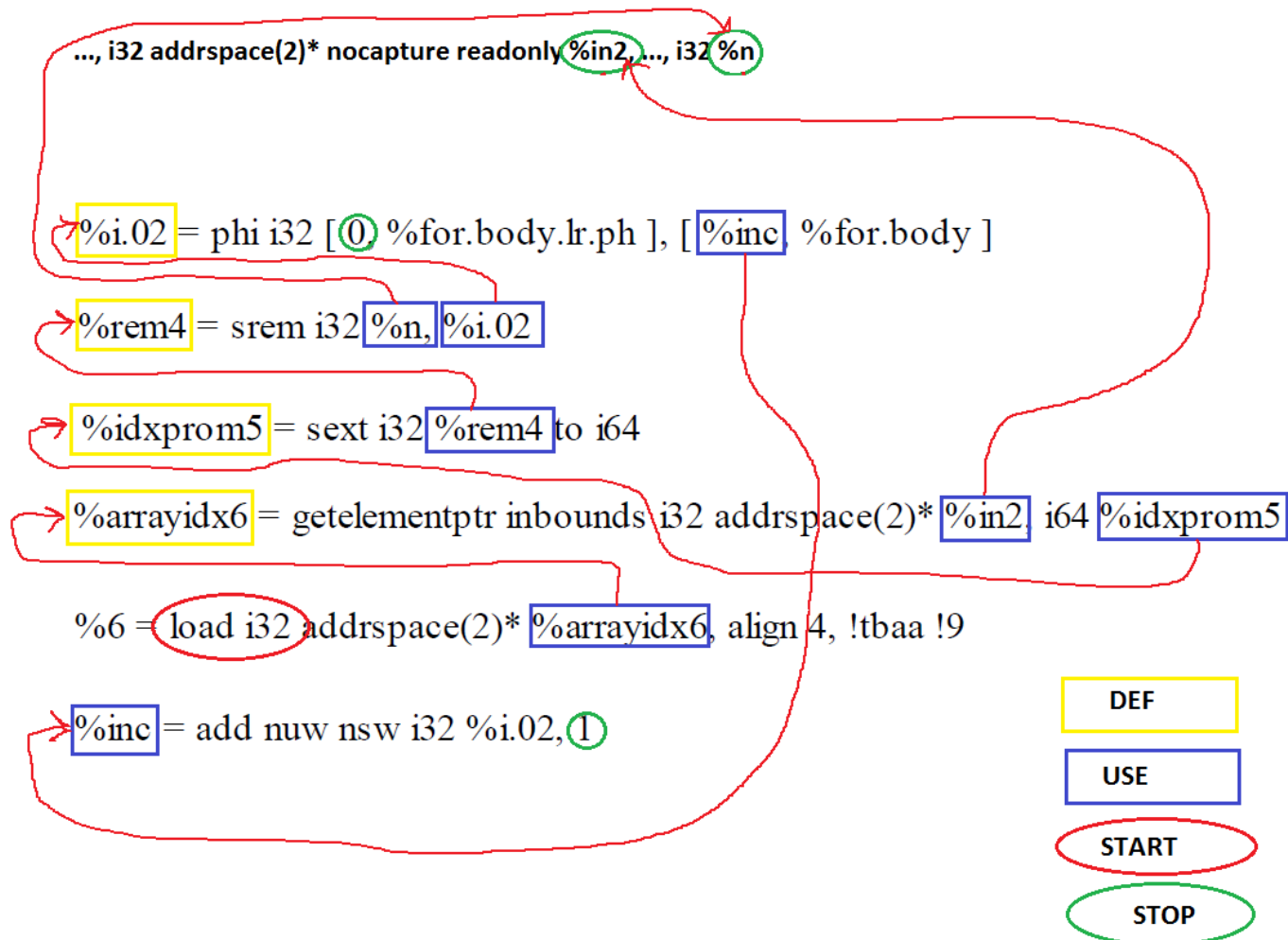
Uniform Slice

```
..., i32 addrspace(2)* nocapture readonly %in2, ..., i32 %n
for.body:
  %lsr.iv1 = phi i32 @rspace(2)* [ %scevgep, %for.body ], [ %in2,
... %for.body.lr.ph ]
  %lsr.iv = phi i32 [ %lsr.iv.next, %for.body ], [ %n, %for.body.lr.ph ]
  %i.02 = phi i32 [ 0, %for.body.lr.ph ], [ %inc, %for.body ]
  %4 = load i32 @rspace(1)* %arrayidx, align 4, !tbaa !9
  %5 = load i32 @rspace(2)* %lsr.iv1, align 4, !tbaa !9
  %rem4 = srem i32 %n, %i.02
  %idxprom5 = sext i32 %rem4 to i64
  %arrayidx6 = getelementptr inbounds i32 @rspace(2)* %in2, i64 %idxprom5
  %6 = load i32 @rspace(2)* %arrayidx6, align 4, !tbaa !9
  %div = sdiv i32 %5, %6
  %add = add nsw i32 %div, %4
  store i32 %add, i32 @rspace(1)* %arrayidx8, align 4, !tbaa !9
  %inc = add nuw nsw i32 %i.02, 1
  %lsr.iv.next = add i32 %lsr.iv, -1
  %scevgep = getelementptr i32 @rspace(2)* %lsr.iv1, i64 1
  %exitcond = icmp eq i32 %lsr.iv.next, 0
  %negate_loop_exit_cond = xor il %exitcond, true
  br il %negate_loop_exit_cond, label %for.body, label %for.end
```



Example

Uniform slice



Example

Slice evaluation

SSA name	%i.02	%idxprom5	%arrayidx6	%inc	%in2	%n	1	0
Width	all	all	all	all	all	all	all	all

- All operations of the slice have width 'All' i.e. are initially uniform
- Analysis will stop at the first iteration

Example

Non uniform slice

**i32 addrspace(1)* nocapture
readonly %in1**

```
entry:  
%0 = tail call spir_func i64 @.ld kernarg_u64(i32 0) #1  
%1 = tail call spir_func i32 @.get_global_id(i32 0) #1  
%2 = zext i32 %1 to i64  
%3 = add i64 %2, %0  
%conv = trunc i64 %3 to i32  
%cmp1 = icmp sgt i32 %n, 0  
br i1 %cmp1, label %for.body.lr.ph, label %for.end
```

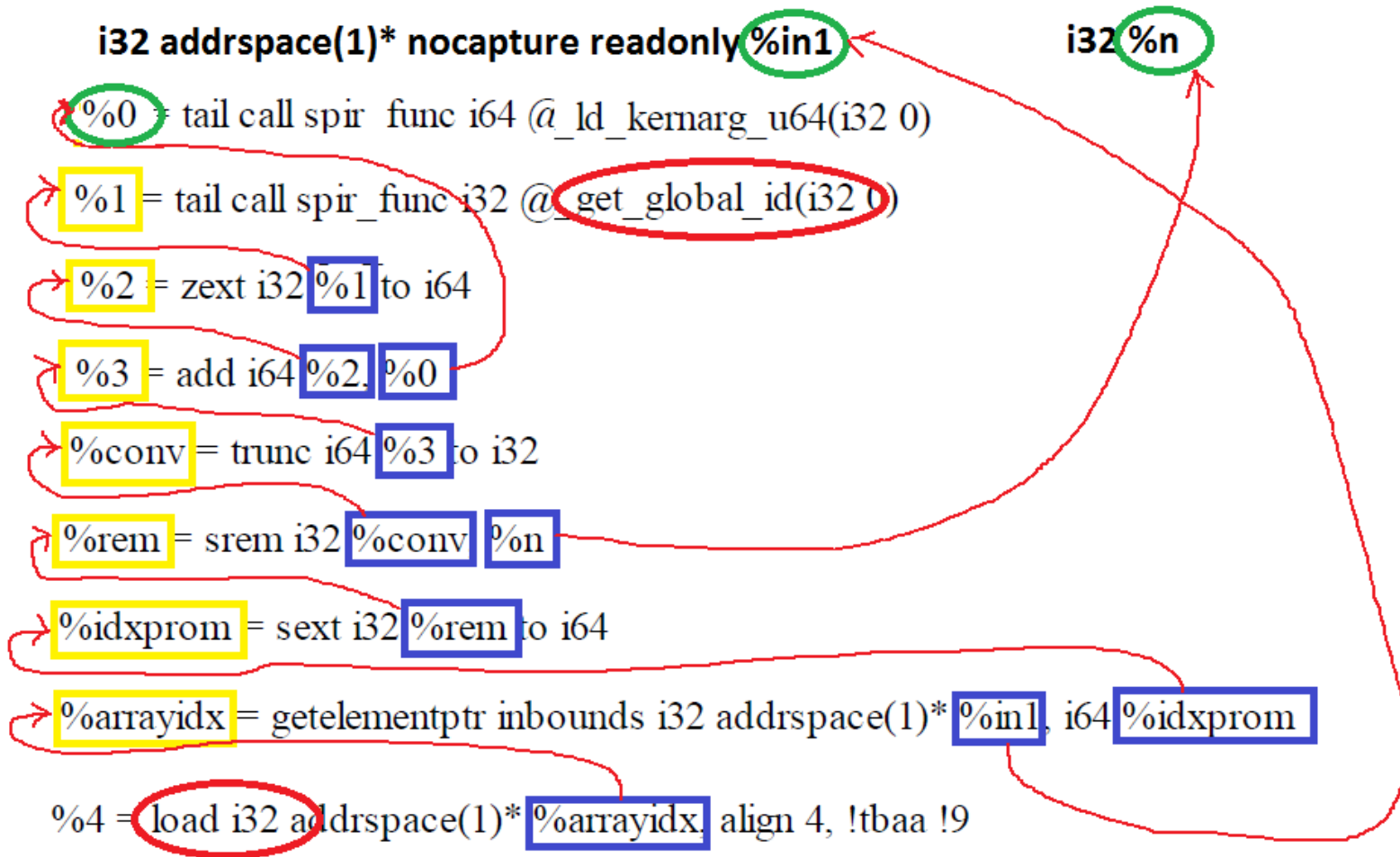
T	F
---	---

```
for.body.lr.ph:  
%rem = srem i32 %conv, %n  
%idxprom = sext i32 %rem to i64  
%arrayidx = getelementptr inbounds i32 @rspace(1)* %in1, i64 %idxprom  
%idxprom7 = sext i32 %conv to i64  
%arrayidx8 = getelementptr inbounds i32 @rspace(1)* %out, i64 %idxprom7  
br label %for.body
```

```
for.body:  
%lsr.iv1 = phi i32 @rspace(2)* [ %scevgep, %for.body ], [ %in2,  
... %for.body.lr.ph ]  
%lsr.iv = phi i32 [ %lsr.iv.next, %for.body ], [ %n, %for.body.lr.ph ]  
%i.02 = phi i32 [ 0, %for.body.lr.ph ], [ %inc, %for.body ]  
%4 = load i32 @rspace(1)* %arrayidx, align 4, !tbaa !9
```

Example

Non uniform slice



Example

Non uniform slice evaluation

SSA name	%arrayidx	%idxprom	%in1	%rem	%conv	%3	%2	%1	%0	%n	get_global_id(0)
width	all	all	all	all	all	all	all	all	all	all	1

$$get_global_id \in E \quad W(get_global_id) = 1$$

SSA name	%arrayidx	%idxprom	%in1	%rem	%conv	%3	%2	%1	%0	%n	get_global_id(0)
Width	All	All	All	All	All	All	all	1	all	all	1

Instructions are processed in order so really the next iteration will be:

SSA name	%arrayidx	%idxprom	%in1	%rem	%conv	%3	%2	%1	%0	%n	get_global_id(0)
Width	1	1	all	1	1	1	1	1	all	all	1

$$W(\%3) = \text{MIN}(W(\%2), W(\%0)) : 1$$

$$W(\%3) = W(\%3) : 1$$

$$W(\%rem) = \text{MIN}(W(\%conv), W(\%n)) : 1 \quad W(\%idxprom) = W(\%rem) : 1$$

$$W(\%arrayidx) = \text{MIN}(W(\%in1), W(\%idxprom)) : 1$$

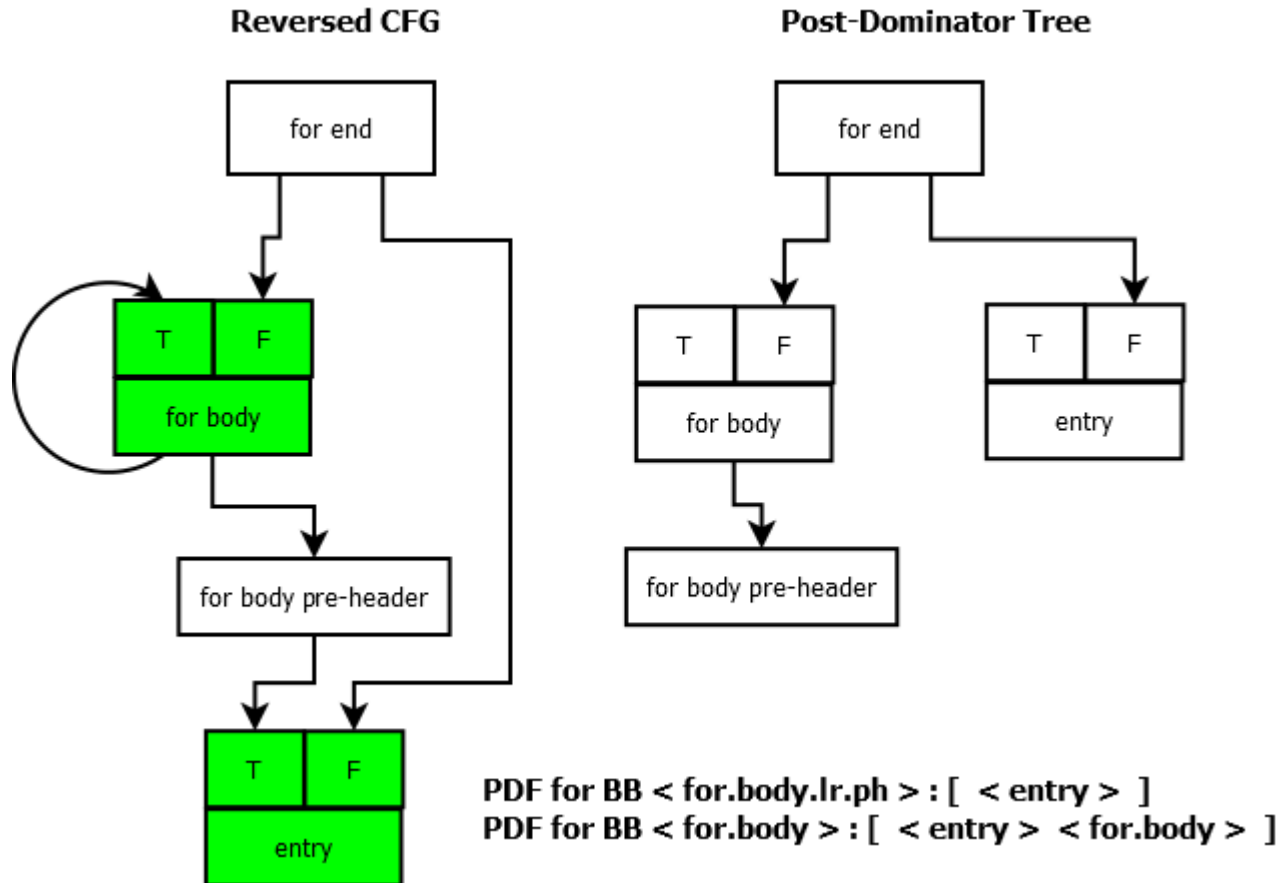
Example

Control dependency

```
__kernel void test(__global int * in, __global int * out, int n)
{
    int idx = 0;
    int tid = get_global_id(0);
    for (int i=0; i<tid; i++) {
        if (i%n)
            idx += i;
    }
    out[0] = in[idx];
}
```

Example

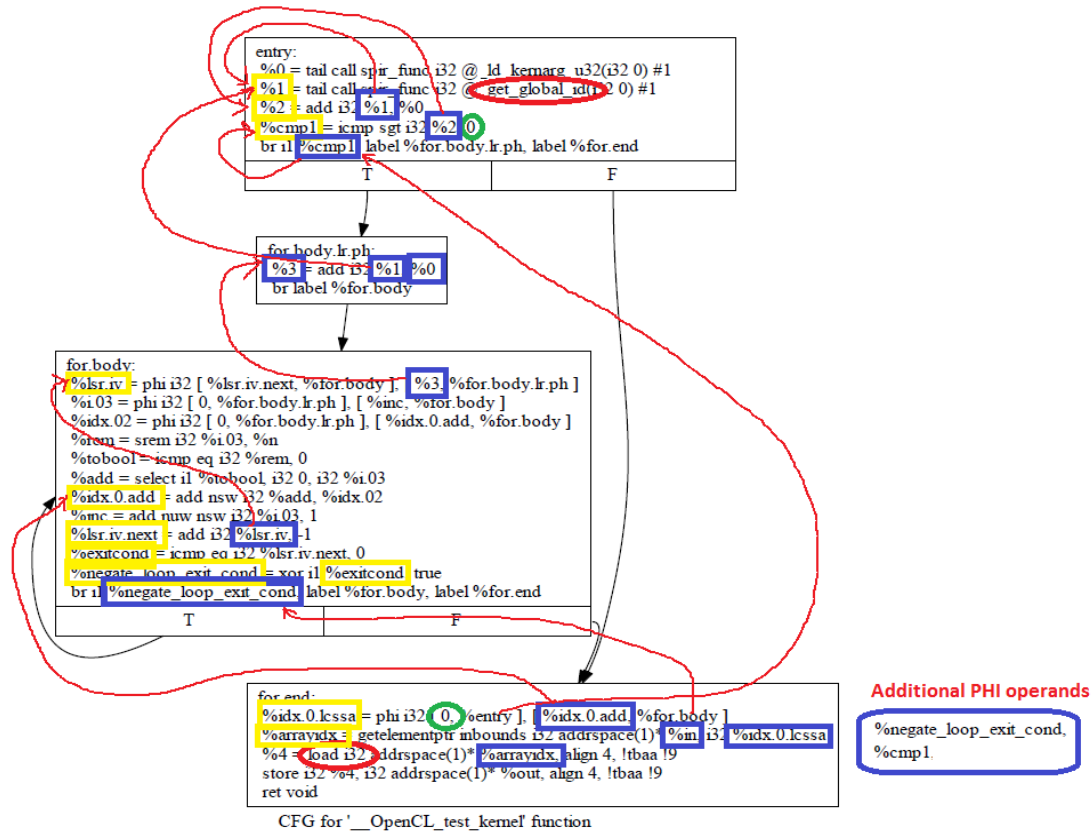
Control dependency



Example

Control dependency

i32 addrspac(1)* nocapture readonly %in, i32 addrspac(1)* nocapture %out, i32 %n



PDF for BB < for.body.lr.ph > : [< entry >]
 PDF for BB < for.body > : [< entry > < for.body >]
 PDF for BB < for.end > : []

CD(%idx.0.lcssa = phi i32 [0, %entry], [%idx.0.add, %for.body]) = T((for.body, entry))

br i1 %negate_loop_exit_cond, label %for.body, label %for.end
 br i1 %cmp1, label %for.body.lr.ph, label %for.end

What it costs and what it yields

- We implement the analysis in AMD OpenCL compiler
- We test the performance on the Radeon R7 GPU
- Performance gain:
 - 10% on HEVC benchmark
 - 3% on Compubench Face Detection test
 - 4% on Video Composition test
- Small overhead:
 - Less than 5% of compile time increase for 20000 lines OpenCL source file

- In HSA compiler – fully employ analysis results in Finalizer
- In AMDGPU compiler – explicitly select vector or scalar form of the instruction depending on the analysis results
- **Is This Upstreamable?**
 - Yes, if the community is interested
 - Yes, if we have a way to legally pass user-defined instruction level metadata to Instruction Selection.