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
Abstract parallel machine: memory

- 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 necessary 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
Sources of non-uniformity

Data dependency

```c
__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

```c
__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
Data dependency analysis

- 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:

\[
W(x) \text{is defined upon the poset } W \text{ such as } \\
W: \{\bot < w_0 < w_1 < \ldots < \top\} \\
W(x) = \bigwedge_{y \in O(x)} W(y) \\
\text{where } O(x) \text{is } x' \text{s operands set} \\
\text{and } \bigwedge\{A\} \text{is MIN over elements of } A
\]

\[\forall x \in W \bot \bigwedge x = \bot, \top \bigwedge x = x\]

\[
W(\text{constant value}) = \top, W(\text{kernel argument}) = \top \\
W(x) = \bot \forall x \in E \text{ where } E \text{ is a set of non-uniform operations}
\]
### Data dependency analysis

- At the analysis start:
  - \( W(x) = 1 \quad \forall x \notin E \quad \text{and} \quad W(x) = \bot \quad \forall x \in E \)

- 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

```cpp
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 $\text{PHI} \subset I$ — set of all $\varphi$ — functions
  - Let $\text{BB} \subset I$ — set of all Basic Blocks of the Function
  - Let $B(i) : i \in I \rightarrow b \in \text{BB}$ 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 $T = \text{set of all terminators and } T(x) : b \in \text{BB} \rightarrow \{t \mid t \in T\}$
  - Let $\text{CD}(\varphi) : \varphi \in \text{PHI} \rightarrow \{i \mid i \in I\}$
  - Let $\text{PDF}(bb) : bb \in \text{BB} \rightarrow \{b \mid b \in \text{BB}\}$

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

PDF(BB8) = \{BB3, BB2\}
PDF(BB1) = \{\}\nPDF(BB5) = \{BB2\}
CD(PHI1) = T(\{BB3, BB2\}\{BB2\}) = T(BB3)

PDF(BB9) = \{BB2\}
PDF(BB6) = \{\}\nCD(PHI2) = T(\{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
__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

```c
entry:
%0 = call void (i64 (i32), [i8*], [i8*], [i8*]) @__hls__lshl_int32.i64(i32) #1
%1 = call void (i64 (i32), [i8*], [i8*], [i8*]) @__hls__lshr_int32.i64(i32) #1
%2 = load i32, i32* %1, align 4, zero beginnings
%3 = add i32 %2, %1
%4 = icmp eq i32 %3, %1
%true = br label %for
%false = br label %for.body

for.body:phi:
%true = phi i32 %2 %false, %false
falseprom = load i32, i32* %true, align 4, zero to i64

for.body:phi:
%false = phi i32 %2 %true, %true
%handle = getelementptr inbounds i32 [i32 adderpace(1)*] %false, i64 %falseprom
%falseprom7 = load i32, i32* %false, align 4, zero to i64
%falseprom8 = getelementptr inbounds i32 [i32 adderpace(1)*] %false, i64 %falseprom7
br label %for.body

for.end:

for.end:ret void
```

CFG for '__OpenCL_test_kernel' function
... i32 addrspace(2)* nocapture readonly %in2, ..., i32 *%n

for.body:
  %lsr.iv1 = phi i32 addrspace(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 addrspace(1)* %arrayidx, align 4, !tbaa !9
  %5 = load i32 addrspace(2)* %lsr.iv1, align 4, !tbaa !9
  %rem4 = srem i32 %on, %i.02
  %idxprom5 = sext i32 %rem4 to i64
  %arrayidx6 = getelementptr inbounds i32 addrspace(2)* %in2, i64 %idxprom5
  %6 = load i32 addrspace(2)* %arrayidx6, align 4, !tbaa !9
  %div = sdiv i32 %5, %6
  %add = add nsw i32 %div, %4
  store i32 %add, i32 addrspace(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 addrspace(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

```assembly
... i32 addrspace(2)* nocapture readonly %in2 ..., i32 %n

%%%i.02 = phi i32 [0 %for.body.lr.ph], [%inc %for.body]

%rem4 = srem i32 %n, %i.02

%idxprom5 = sext i32 %rem4 to i64

%arrayidx6 = getelementptr inbounds i32 addrspace(2)* %in2 i64 %idxprom5

%6 = load i32 addrspace(2)* %arrayidx6 align 4, !tbaa !9

%inc = add nuw nsw i32 %i.02, 1
```

### Example

**Slice evaluation**

<table>
<thead>
<tr>
<th>SSA name</th>
<th>%i.02</th>
<th>%idxprom5</th>
<th>%arrayidx6</th>
<th>%inc</th>
<th>%in2</th>
<th>%n</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
</tbody>
</table>

- 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 @_kld_kernarg_u64(i32 0) #1
  %1 = tail call spir_func i32 @get_global_id(i32 0) #1
  %2 = sext i32 %1 to i64
  %3 = add i64 %2 %0
  %conv = trunc i64 %3 to i32
  %cmp1 = icmp sgt i32 %n, 0
  br if %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 addrspace(1)* %in1, i64 %idxprom
  %idxprom7 = sext i32 %conv to i64
  %arrayidx8 = getelementptr inbounds i32 addrspace(1)* %out, i64 %idxprom7
  br label %for.body
```

```
for.body:
  %slr.iv1 = phi i32 addrspace(2)* [%scevgep, %for.body ], [ %in2, ...
  %for.body.lr.ph ]
  %slr.iv = phi i32 [%slr.iv.next, %for.body ], [%n, %for.body.lr.ph ]
  %i02 = phi i32 [ 0, %for.body.lr.ph ], [%inc, %for.body ]
  %4 = load i32 addrspace(1)* %arrayidx, align 4, !tbaa !9
```
Example
Non uniform slice

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

%%0 = tail call spir_func i64 @ld_kernarg_u64(i32 0)

%%1 = tail call spir_func i32 @get_global_id(i32 0)

%%2 = cast i32 %%1 to i64

%%3 = add i64 %%2, %%0

%%%conv = trunc i64 %%3 to i32

%%rem = srem i32 %%conv %%n

%%idxprom = cast i32 %%rem to i64

%arrayidx = getelementptr inbounds i32 addrspace(1)* %in1, i64 %%idxprom

%%4 = load i32 addrspace(1)* %arrayidx align 4, !tbaa !9
```
Example
Non uniform slice evaluation

<table>
<thead>
<tr>
<th>SSA name</th>
<th>%arrayidx</th>
<th>%idxprom</th>
<th>%in1</th>
<th>%rem</th>
<th>%conv</th>
<th>%3</th>
<th>%2</th>
<th>%1</th>
<th>%0</th>
<th>%n</th>
<th>get_global_id(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
</tbody>
</table>

\[
global_id \in E \quad W(global_id) = 1
\]

<table>
<thead>
<tr>
<th>SSA name</th>
<th>%arrayidx</th>
<th>%idxprom</th>
<th>%in1</th>
<th>%rem</th>
<th>%conv</th>
<th>%3</th>
<th>%2</th>
<th>%1</th>
<th>%0</th>
<th>%n</th>
<th>get_global_id(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>all</td>
<td>1</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SSA name</th>
<th>%arrayidx</th>
<th>%idxprom</th>
<th>%in1</th>
<th>%rem</th>
<th>%conv</th>
<th>%3</th>
<th>%2</th>
<th>%1</th>
<th>%0</th>
<th>%n</th>
<th>get_global_id(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>1</td>
<td>1</td>
<td>all</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>all</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
W(\%3) = \text{MIN}(W(\%2), W(\%0)) : 1 \quad W(\%3) = W(\%3) : 1
\]
\[
W(\%\text{rem}) = \text{MIN}(W(\%\text{conv}), W(\%n)) : 1 \quad W(\%\text{idxprom}) = W(\%\text{rem}) : 1
\]
\[
W(\%\text{arrayidx}) = \text{MIN}(W(\%\text{in1}), W(\%\text{idxprom})) : 1
\]
Example
Control dependency

```c
__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

Reversed CFG

for end

for body

for body pre-header

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

Post-Dominator Tree

for end

for body

for body pre-header

tenry
Example
Control dependency

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

```
entry:

add i32 %1, %0, %0
store addrspace(1)*, i32 %2, %1, align 4

if cmp %l, %0, %0
br i1 %cbr1
```

```
for_body:

add i32 %1, %2, %2
br label %for_body
```

```
for_end:

%negate_loop_exit cond = xor i1 %truecond true
br i1 %negate_loop_exit cond label %for body, label %for end
```

Additional PHI operands

```
%negate_loop_exit cond, %cbr1.
```

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

CD( %idx 0.kcssa = phi i32 [0, %entrv 1, %idx 0.add, %for.body ] = T((for.body, entry))
br i1 %negate_loop_exit cond, label %for body, label %for end
br i1 %cbr1 label %for body & 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
Future work

- 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.