Code-Generation for the Arm M-profile Vector Extension

Sjoerd Meijer
Sam Parker
David Green

Sjoerd.meijer@arm.com, sam.parker@arm.com, david.green@arm.com

US LLVM Developer conference 2019
Major MVE Architecture Features

- Low-overhead branches, a.k.a. hardware-loops
  - Improved performance for the loop test, increment, branch

- Vectorisation
  - 128-bit vector size
  - 8 vector registers
  - 1 predicate register

- Predication:
  - Tail-loop predication
  - Lane-predication, a.k.a. VPT predication (like IF-THEN blocks)
The Challenge is....

All these features can be combined!

"Tail-predicated and vectorised hardware-loop"

1. What is it?

2. Why is it useful?

3. How should the transformation pipeline look like to achieve this?
# Predication

1. **Tail-loop predication**
   - Removes the need for generating a scalar remainder loop.

2. **Lane / VPT Predication**

<table>
<thead>
<tr>
<th>Scalar code example</th>
<th>Vector pseudo code</th>
<th>VPT Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i= 0; i &lt; VL; i++)</td>
<td>mask = A &lt; B</td>
<td>VPTE.s32</td>
</tr>
<tr>
<td>if (A[i] &lt; B[i])</td>
<td>T: A = B + C</td>
<td>lt, q0, q1</td>
</tr>
<tr>
<td>else</td>
<td></td>
<td>q0, q1, q2</td>
</tr>
<tr>
<td>A[i] = B[i] + D[i]</td>
<td></td>
<td>VADDE.i32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>q0, q1, q3</td>
</tr>
</tbody>
</table>
Tail-Predication: A Code Example

```c
void foo(int *A, int *B, int *C, int N)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

- Loop is vectorisable
  - Assume restrict, or runtime checks
- Unknown trip count N:
  - Vector body
  - Scalar remainder
Tail-Predication: What Is It?

// Assume N = 10, VF = 4

// 1. vector body
for (int i = 0; i <= 8; i+=4)

// 2. scalar remainder: the tail
for (i = 0; i < 2; i++)
    C[i] = A[i] + B[i];

// 3. tail-folded vector loop
for (int i = 0; i < 12; i+=4)
    M = get_mask(i:4, N);

---

vector loop iterations:

- **i = 0**
  - 0, 1, 2, 3

- **i = 4**
  - 4, 5, 6, 7

- **i = 8**
  - 8, 9, 10, 11

---

4 lanes:
- **Active lanes**
- **Inactive lanes**
Tail-Predicated Vector Hardware-Loop: Why is it useful?

Tail-predicated vector hardware-loop:
• **4 instructions** for the loop-body:
• **2 instructions** for controlling the loop(s)

Benefits:
1. **Code-size**: important for microcontrollers.
2. **Performance**: data sets relatively small, overhead of the tail can be significant.
Typical Vectorisation Plan:

```c
if (N == 0) goto END;

if (N < 4)
    goto REMAINDER_LOOP;
else
    goto VEC_LOOP;

VEC_LOOP:
    for (..)

REMAINDER_LOOP:
    for (..)
        C[i] = A[i] + B[i];

END:
    return;
```

Vectorisation and Predication:

- How do we create 1 vector loop that combines the vector and scalar code?

Hardware-loops:

- Do we create hwloops before, during, or after vectorisation?
- How is a hwloop represented in IR, and how is it lowered?
Optimisation Pipeline & Contributions

1. Loop vectoriser
2. Tail-folded vector loop
3. IR Tail Pred Pass
4. Tail-folded vector hwloop

- Scalar Loop
- Loop vectoriser
- Tail-folded vector loop
- IR hardware Loop pass

- ARM Backend
- Tail-predicate vector hwloop
- IR Tail Pred Pass
- Tail-folded vector hwloop
wlstp.32  lr, r3, END
.LBB0_4:
vldrw.u32 q0, [r0, #16]!
vldrw.u32 q1, [r1, #16]!
vadd.i32 q0, q1, q0
vstrw.32 q0, [r2, #16]!
letp  lr, .LBB0_4

END:

Optimisation Pipeline - Backend

ISEL & Pseudo exp.
Loop
Low-overhead loop & Tail-predication (todo)

4

Revert if necessary: hwloops, Tail-predication
arm

Transformation Details
1. Vectorisation: Tail Folding

**Pseudo code**

```c
vector.preheader:
    Vimax = (N, N, N, N)

vector.body:
    Vinit = (i, i, i, i)
    Vinc = (0, 1, 2, 3)
    Vi = Vinit + Vinc
    ActiveLaneMask = Vi < Vimax
    i += 4
```

**Enabled in multiple ways**

- #pragma clang loop vectorize_predicate(enable)
- -prefer-predicate-over-epilog=true
- Optimise for size.
- Querying a TTI hook (in progress).
2. Hardware Loops

- A generic LLVM IR LoopPass which inserts intrinsics into loops, designating it as a 'hardware loop'.
- Transforms natural loops with a known trip count.
- Works on both vector and scalar loops.
- TargetTransformInfo hook to check support and profitability of the conversion.
  - Also conveys target details back to the pass which enables configurability.
- Based upon the PPCCTRLoops pass.
- Now used by both PPC and ARM backends.
2. Hardware Loop: Intrinsics

- `void llvm.set.loop.iterations(anyint)`
  - Set the counter.

- `i1 llvm.test.set.loop.iterations(anyint)`
  - Set and test that the given operand is not zero.

- `i1 llvm.loop.decrement(anyint)`
  - Signal whether to continue looping.

- `anyint llvm.loop.decrement.reg(anyint, anyint)`
  - A fancy sub for the iteration count.
2. Hardware Loops: Example

preheader:

```plaintext
call void @llvm.set.loop.iterations(i32 %num.iterations)
br label %body
```

body:

```plaintext
%count = phi i32 [ %num.iterations, %preheader ], [ %remaining, %body ]
...
%remaining = call i32 @llvm.decrement.reg(i32 %count, i32 1)
%not.finished = icmp ne i32 %remaining, 0
br i1 %not.finished, label %body, label %exit
```
2. Hardware Loops: Guard entry

```assembly
entry:
  %non.zero = call i1 @llvm.test.set.loop.iterations(i32 %num.iterations)
  br i1 %non.zero, label %preheader, label %exit

preheader:
  br label %body

body:
  %count = phi i32 [ %num.iterations, %preheader ], [ %remaining, %body ]
  ...
  %remaining = call i32 @llvm.decrement.reg(i32 %count, i32 1)
  %not.finished = icmp ne i32 %remaining, 0
  br i1 %not.finished, label %body, label %exit
```
3. MVE Tail Predication

- An IR LoopPass which inserts Arm specific intrinsics to perform vector predication.
- Find hardware loop intrinsics.
- Find masked load/store intrinsics.
- Masked load/store intrinsics take a lane predicate.
- Replace lane masking vector icmp sequence with an intrinsic.
3. VCTP 'Vector Create Tail Predicate'

- Vectorizer will generate a specific pattern for masking out inactive lanes.
- The loop invariant 'broadcast.splat.limit' gives us our total number of elements that we need to process.
- Given the number of elements to process and the effective vector width, generate a mask for the inactive lanes.

```assembly
%index = phi i32
%broadcast.splatinsert = insertelement <4 x i32> undef, i32 %index, i32 0
%broadcast.splat = shufflevector <4 x i32> %broadcast.splatinsert,
               <4 x i32> undef,
               <4 x i32> zeroinitializer
%induction = add <4 x i32> %broadcast.splat, <i32 0, i32 1, i32 2, i32 3>
%pred = icmp ule <4 x i32> %induction, %broadcast.splat.limit
%load = tail call <4 x i32> @llvm.masked.load..<4 x i1> %pred...
```
3. Replacing icmp sequence with vctp

preheader:
  call void @llvm.set.loop.iterations(i32 %num.iterations)
  br label %vector.body

vector.body:
  %count = phi i32 [ %num_iterations %preheader ], [ %count.next %vector.body ]
  %elems = phi i32 [ %total.elems %preheader ], [ %elems.remaining %vector.body ]
  %tail.pred = call <4 x i1> @llvm.arm.vctp32(i32 %elems)
  %elems.remaining = sub i32 %elems, 4
  ...
  %count.next = call i32 @llvm.decrement.reg(i32 %count, i32 1)
  %not.finished = icmp ne i32 %count.next, 0
  br i1 %not.finished, label %vector.body, label %exit
4. Instruction Selection

- `llvm.set.loop.iterations` → `DoLoopStart (Pseudo)`
- `llvm.test.set.loop.iterations` → `WhileLoopStart (Pseudo)`
- `br`
- `llvm.arm.vctp` → 1-1 mapping with real instruction
- `llvm.loop.decrement.reg icmp ne/eq/ugt, etc... br`
- `Separate the decrement from the control flow:
  - LoopDec (Pseudo)
  - LoopEnd (Pseudo)`

Need to check branch targets and conditions when combining nodes!
4. Pre-emission Finalisation

• Right at the end of the pipeline.
• Need to handle scalar and vector loops (predicated or not).
• Either generate special loop instructions or revert to sub, compare and branch.
  • DoLoopStart = DLS, DSLTP
  • WhileLoopStart = WLS, WLSTP
  • LoopDec, LoopEnd = LE, LETP
• Expand pseudo instructions.
  • Their sizes are large enough to compensate for extra instructions that maybe inserted.
• Convert explicit predication to implicit predication (which we haven't done yet...)
4. When do we revert to a 'normal' loop?

- If `LoopDec` is used by another instruction other than `LoopEnd`.
  - Loop counter is held in the link register (LR), which is available to the register allocator.
  - The 'LE' instruction will be performing the decrement and the branch, so the real value of `LoopDec` does not exist before the terminator!
- If we don’t discover all the necessary pseudo instructions.
- Branch targets need to be in range:
  - `LoopStart` instructions can only branch forwards.
  - `LoopEnd` can only branch backwards.
- No calls within the loop.
  - Though this is not an architectural requirement.
4. When won't we convert to implicit tail predication?

- Finalising after VPT (predicated vector) blocks have been generated.
- We can't use implicit predication if we don't know how to remove the existing VPT blocks.
- If we don't understand the predicate dataflow:
  - VPT blocks can represent nested conditional statements.
  - We have ONE predicate register which can be written, spilled, reloaded, etc...
  - Need dataflow analysis.
- If we don’t know how to tail predicate an instruction.
  - This is a temporary lack of semantic understanding on our part.
Where we are, where we want to be

Explicit: LR holds an iteration count while vctp operates on another register holding an element count.

```
.wls lr,
.body:
vctp.32 ; Generate Predicate
sub
vpstt ; VPT block
vldrwt.u32 ; Predicated load
vldrwt.u32 ; Predicated load
vadd.i32
vpst ; VPT block
vstrwt.32 ; Predicated store
le lr, .body
```

Implicit: LR holds the element count and predication is performed via the loop start/end instructions via a system register.

```
wlstp.32 lr, ; Generate Predicate
.body:
vldrw.u32 ; Predicated load
vldrw.u32 ; Predicated load
vadd.i32 ; Predicated add
vstrw.32 ; Predicated store
letp lr, .body ; Generate Predicate
```

Smaller, faster, better!
Conclusion

• Created a multi-stage optimisation pipeline to enable tail predicated vector loops.
• Code generation for MVE is in good shape and still improving.
• Development has been relatively quick because we were able to reuse existing code.
• Interested in talking to others who rely on analysis and transforms on post-RA machine code.
• Thank you for all your feedback and reviews!