Report on LLVM ShuffleVector Optimizer

ShengZhang LAI

June 25, 2014

Date Performed:	June 17, 2014
Partners:	LAI ShengZhang
	ZHANG LiQian
	LI WanPeng

1 Objective

1.1 Background

Shufflevector is a powerful instruction in LLVM. As Listing 1, the 'shufflevector' instruction constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask(LLVM [2014]).

Listing 1: ShuffleVector Syntax

```
; yields <m x <ty>>
<result> = shufflevector <n x <ty>> <v1>, <n x <ty>> <v2>, <m x i32> <mask>
```

IR code is the middle code for LLVM, which is target independent and transplantable, so in IR level, we can do much optimization based on common instruction set. With more information on the specific target, backend code generator will translate IR code into assembly code for the specific target or instruction set. If one instruction rely on the specific target, such like PEXT and PACK instructions on Intel's architecture, such optimization will be implemented in the backend code generation.

1.2 Goals

Our goal is to create a systematic infrastructure for pattern analysis and to automatically replace the specific shuffle vector instruction with the preferred simple instructions.

2 Experiments

We mainly focus on the 7 patterns as listing below:

a. ROTATE Pattern

b. SHIFT LEFT Pattern

c. LOGICAL SHIFT RIGHT Pattern

d. ARITHMETIC SHIFT RIGHT Pattern

e. BLEND Pattern

f. ZERO-EXTEND Pattern

g. PEXT/PDEP Pattern

Those patterns can be optimized by two means, optimizing in IR level or in backend code generation. In the IR level, an optimizing pass will traverse every IR instruction and replace the specific instruction with the preferred instruction, such like shift left or byte swap. In the backend code generation phase, compiler collects more information on the target. So the backend code generator can do more optimization for the specific instruction set or processor.

2.1 IR Optimization

2.1.1 Methods

In llvm-IR level, we may write a LLVM Pass to detect the existence of ShuffleVector instructions and check whether it can be replaced according to its operands and masks, for better performance.

The whole process we experienced can be divided into two phases: in the first period, check and compare the performance before and after substitution. In the second period, since Zhang LiQian is interested in the recognition of kinds of patterns in the mask of ShuffleVector instruction (operands are also involved in recognition), he works on the first level of optimization as mentioned — the llvm-IR level, to make a systematic approach to detect the pattern lying in the mask, and then to replace the instruction.

In the first period, we examined the performance with two examples in the website, the ShuffleVector instructions that can be replaced by llvm.bswap and SIMDly-done logical bit operations (i.e.shl, lshr), respectively. Firstly, we got to generally be familiar with the IR instructions and manually wrote some benchmarks for testing. Afterwards we first write C-code framework, which is much efficient and clear. Our main goal is to construct a context where there is a ShuffleVector instruction that can be replaced with llvm.bswap for example, with this block in a loop for 10000 times. Then used command 'perf -stat' as a measure. Unfortunately the time-measure result in either the bswap example or the bit operation example is not improved. But we can actually reduce the generated assembly size. So it's worthwhile to do the replacement in some regular

patterns. We think this period is a good start and helps me get to the IR and LLVM infrastructure. We did some instruction replacement in this period and it helps me understand how to replace ShuffleVector instruction with certain mask.

In the second period, Zhang started to work on the ShufflePatternLibrary. Before Zhang study the contents from website by Professor Cameron(Cameron [2014]), he found that an efficient way to recognize the pattern is to differentiate the mask and evaluate the difference, which coincides with the diff and signature functions in the website. He tried to use a concept like "continuous block" to approach it. For example, mask (1, 2, 3, 0) yields to difference (1, 1, -3). The pass receives continuous 1 as continuous block, when find -3, check the number of 1s, and the length is 2. So the pass receive -3 as a circular end of 1s. Similarly -4 is the end of 3 1s. The dual circumstance is (-1, -1, 3) which can be handled in the same way. But the situation became more and more complex.

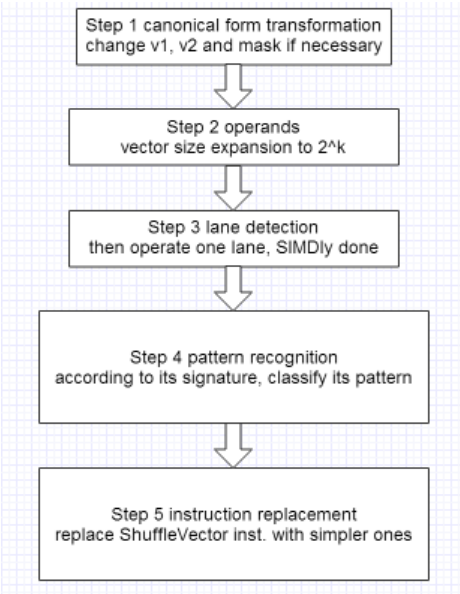

Figure 1: 5 steps in IR optimization

We arrange the process in the website in 5 steps as shown in Figure 1: Step 1 canonical form transformation; Step 2 vector size expansion to 2^k ; Step 3 lane detection; Step 4 pattern recognition; Step 5 instruction replacement. The first three steps we strictly realize the function in accordance with the website, including the rule to transpose the position of v1 and v2 in step1, the rule to extend the size of operands, and the rule to recursively detect the lane number.

In step 4, since we mainly deal with the front level at llvm-IR level. So we implement the detection of patterns in term of the rotate pattern, the shift left pattern, the shift right logical pattern, the shift right arithmetic pattern, the

blend pattern and zero-extend pattern. A class pattern_rec was to recognize the pattern. It includes some basic helpful scalable functions, which can be easily extended to recognize new patterns. For example, count_number function count the occurrence of certain eigenvalue in the mask (for example 1, n-1), new pattern will also have its eigenvalue. In this way we turn the process to input its eighenvalue and signature (namely the difference of the mask) and check the result.

In step 5, we implemented two kinds of replacement, the shift left pattern and the blend pattern. We wrote a class instruction_rep to realize this functionality. It accepts the pattern recognition results in step 4 as input and change the instruction environment around the ShuffleVector instruction by LLVMs API.

In Figure 2, there are some examples to show how the specific shufflevector IR code transforms into the preferred IR operation.

2.1.2 Results

The Figure 3, Figure 4 and Figure 5 shows the part of situations we can handle.

2.2 Haswell instructions

Two new instructions, pext and pdep, has been introduced into Intel's Haswell architecture, which will equip the processor with stronger bit operation ability.

PEXT is short for Parallel Bits Extract. PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed(Intel [2011]).

2.2.1 Methods

Those two instructions operate on 32-bit or 64-bit general register, but vector types v32i1 and v64i1 are not the legal type in current LLVM system. So we need to legalize those types firstly. Take the v32i1 type for instance.

Listing 2: Add vector register for v32i1

```
//In File X86RegisterInfo.td
def VR32: RegisterClass<"X86", [v32i1, i32], 32, (add GR32)>;

//In File X86CallingConv.td
//Vector type v32i1 is stored in GR32Reg.
CCIfType<[v32i1], CCAssignToReg<[EAX, EDX, ECX]>>

//Vector type v32i1 is stored in GR32Reg.
CCIfType<[v32i1], CCAssignToReg<[EDI, ESI, EDX, ECX, R8D, R9D]>>
```

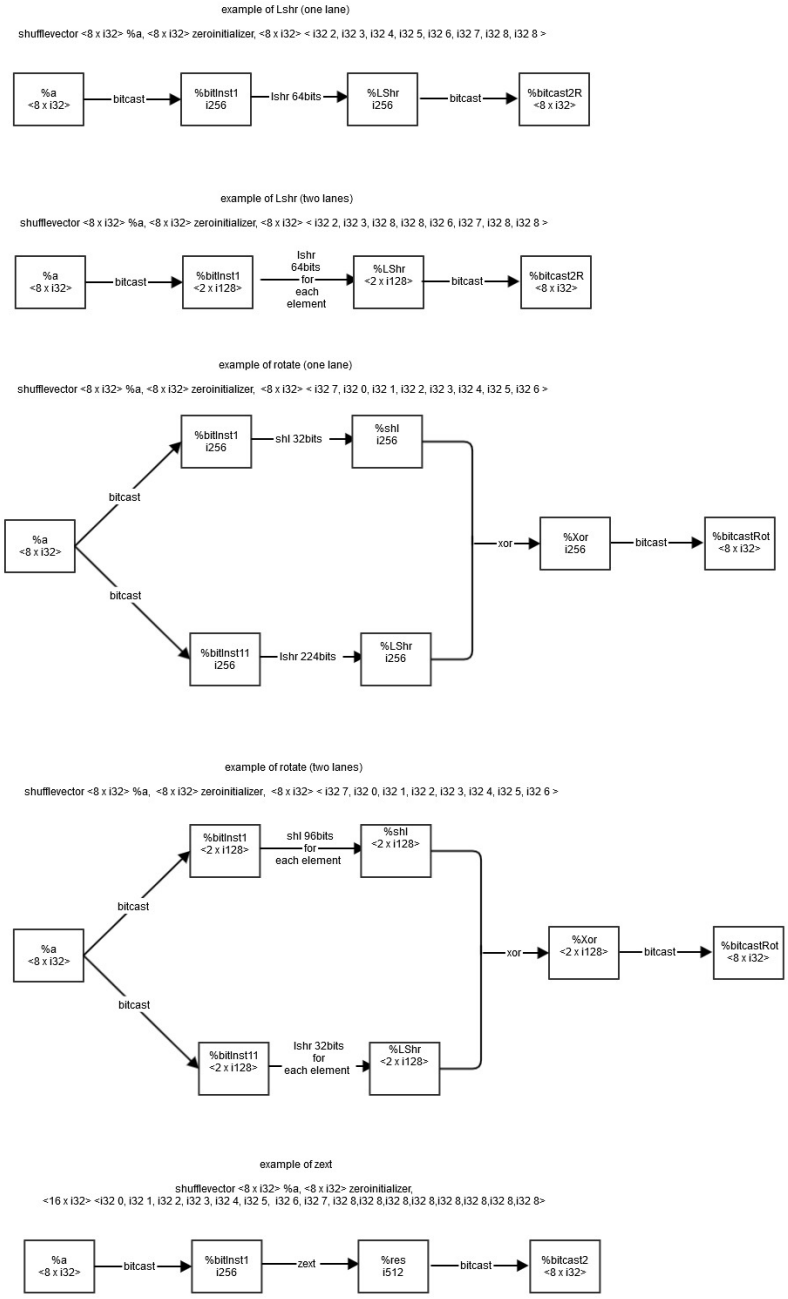

Figure 2: Examples in IR optimization

```
define <8 x i4> @mysfl(<8 x i4> %a, <8 x i4>%b)
{
  %t = shufflevector <8 x i4> %a, <8 x i4> zeroinitializer,
     <8 x i32> <i32 8, i32 0, i32 1, i32 2, i32 8, i32 4, i32 5, i32 6>

  ret <8 x i4> %t
}
```

(a) Input Example

```
define <8 x i4> @mysfl(<8 x i4> %a, <8 x i4> %b) {
  %bcInst1 = bitcast <8 x i4> %a to <2 x i16>
  %shl1 = shl <2 x i16> %bcInst1, <i16 4, i16 4>
  %bitcast2 = bitcast <2 x i16> %shl1 to <8 x i4>
  ret <8 x i4> %bitcast2
}
```

(b) Output Example

Figure 3: Shift Left pattern with 2-lane

```
define <4 x i32> @mysfl(<4 x i32> %a, <4 x i32>%b)
{
  %t = shufflevector <4 x i32> %a, <4 x i32> %b, <4 x i32> <i32 0, i32 5, i32 6, i32 3>
  ret <4 x i32> %t
}
```

(a) Input Example

```
define <4 x i32> @mysfl(<4 x i32> %a, <4 x i32> %b) {
  %slInst1 = select <4 x i1> <i1 true, i1 false, i1 false, i1 true>, <4 x i32> %a, <4 x i32> %b
  ret <4 x i32> %slInst1
}
```

(b) Output Example

Figure 4: Blend pattern

rot-1-lane-2
[0,1,2,3,4,5,6,7] --> [3,0,1,2,7,4,5,6]

(a) Input Example

res-rot-1-lane-1.ll | res-rot-1-lane-2.ll

```
1 ; ModuleID = 'test-rotate/result/res-rot-1-lane-2.bc'
2
3 define <8 x i32> @try(<8 x i32> %a) {
4   %bcInst1 = bitcast <8 x i32> %a to <2 x i128>
5   %shl1 = shl <2 x i128> %bcInst1, <i128 32, i128 32>
6   %bcInst11 = bitcast <8 x i32> %a to <2 x i128>
7   %LShr = lshr <2 x i128> %bcInst11, <i128 96, i128 96>
8   %Xor = xor <2 x i128> %shl1, %LShr
9   %bitcastRot = bitcast <2 x i128> %Xor to <8 x i32>
10  ret <8 x i32> %bitcastRot
11 }
```

(b) Output Example

Figure 5: Rotate pattern with 2-lane

By adding those code in listing 2 into corresponding files, we associate the 32-bit vector type v32i1 with the 32-bit Generic Registers. And the code like in the listing 3 should be added into function X86TargetLowering::resetOperationActions of file X86ISelLowering.cpp, which protect v32i1 type from promoting or extending into other types.

Listing 3: Legalize type v32i1

```
//In File X86ISelLowering.cpp
//Function: X86TargetLowering::resetOperationActions()
addRegisterClass(MVT::v32i1, &X86::VR32RegClass);
```

Listing 4: Set action for VectorShuffle with Type v32i1

```
//In File X86ISelLowering.cpp
//Function: X86TargetLowering::resetOperationActions()
setOperationAction(ISD::VECTOR_SHUFFLE, MVT::v32i1, Custom);

setOperationAction(ISD::BITCAST, MVT::v32i1, Legal);

//In File X86InstrInfo.td
def : Pat <(i32 (bitconvert (v32i1 VR32:$src))), (i32 VR32:$src)>;
```

The second step, code in listing 4 makes function LowerVECTOR_SHUFFLE to take the charge of lowering generation for SDNode VECTOR_SHUFFLE with type v32i1. And we need to make 'bitcast' from v32i1 to i32 legal by staying in the same register.

Listing 5: isPEXTMask and getPEXT

```
//In File X86ISelLowering.cpp

// isPEXTMask - Return true if the specified VECTOR_SHUFFLE operand
// specifies a shuffle of elements that is suitable for input to PEXT
// in haswell architecture.
static bool isPEXTMask(ArrayRef<int> Mask, MVT VT,
                       SDValue V1, SDValue V2) {
    ......
}

// getPEXT - Returns a PEXT node for an pext operation.
static SDValue getPEXT(SelectionDAG &DAG, SDLoc dl, EVT VT, SDValue V1,
                       SDValue V2, ArrayRef<int> Mask) {
    ......
}
```

In the third step, we implement two functions as listing 5 for 'PEXT' instruction, isPEXTMask and getPEXT. 'isPEXTMask' is to check the mask of VECTOR_SHUFFLE, and it returns true when this VECTOR_SHUFFLE can be transformed into PEXT. 'getPEXT' is to replace the VECTOR_SHUFFLE SDNode with the corresponding PEXT SDNode once 'isPEXTMask' returns true. In the end, in the function LowerVECTOR_SHUFFLE, we can use the two functions to check and replace the specified VECTOR_SHUFFLE SDNode.

2.2.2 Results

Take PEXT Pattern Optimization for instance. IR code in Listing 6 can be transformmed into 'PEXT' SDNode in Figure 6, then into the Assembly code in Listing 7. By this transformation, shufflevector will take the advantages of 'PEXT'.

Listing 6: Input of PEXT Pattern Optimization

```
define <32 x i1> @mypext(<32 x i1> %a) {
    %p = shufflevector <32 x i1> %a, <32 x i1> zeroinitializer,
                       <32 x i32>
          <i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32,
           i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32,
           i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32, i32 32,
           i32 0, i32 4, i32 7, i32 8, i32 9, i32 10, i32 15, i32 20>
    ret <32 x i1> %p
}
```

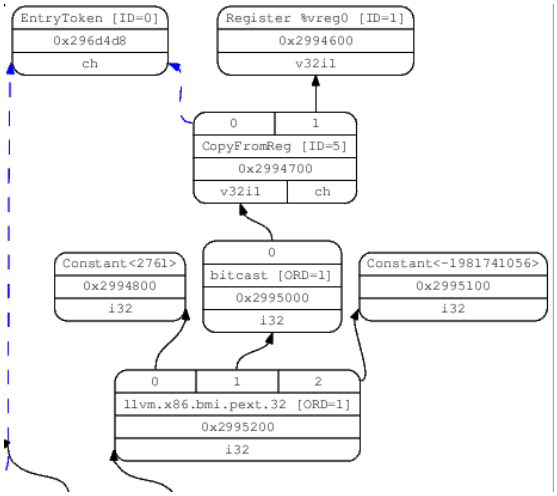

Figure 6: PEXT SDNode

Listing 7: Output of PEXT Pattern Optimization

```
        .file   "/tmp/shuffle.ll"
        .text
        .globl  mypext
        .align  16, 0x90
        .type   mypext,@function
mypext:                                 # @mypext
        .cfi_startproc
# BB#0:
        movl    $-1981741056, %eax      # imm = 0xFFFFFFFF89E10800
```

```
        pextl   %eax, %edi, %eax
        ret
.Ltmp0:
        .size   mypext, .Ltmp0-mypext
        .cfi_endproc

        .section        ".note.GNU-stack","",@progbits
```

3 Further Work

ShuffleVector is a powerful instruction in LLVM language. With new instructions will be introduced into new architecture, we can use these interesting instructions to implement or optimize LLVM code generation.

This project is just a demo shows how we can optimize for shufflevector for LLVM. There is much work to be done before we put such above mentioned optimization into real use. Like with v32i1 vector type, we need to let backend code generation still work when the shufflevector with v32i1 can not be transformmed into 'PEXT'.

References

R. Cameron. Towards a library of shufflevector patterns. http://parabix.costar.sfu.ca/wiki/ShufflePatternLibrary, 2014.

Intel. Intel Advanced Vector Extensions Programming Reference. https://software.intel.com/sites/default/files/m/8/a/1/8/4/36945-319433-011.pdf, 2011.

LLVM. Llvm language reference manual. http://www.llvm.org/docs/LangRef.html#shufflevector-instruction, 2014.