Integer Vector Optimizations and "Usual Arithmetic Conversions"

Stephen Rogers (stephen.rogers@movidius.com)

Martin O'Riordan (martin.oriordan@movidius.com)

ABSTRACT

ISO C and ISO C++ evaluate arithmetic expressions using a common type. The common types are determined by the "Usual Arithmetic Conversions" [UAC]. While these are necessary to ensure that the Standards are properly implemented, they have significant implications for vectorizing optimizations.

This paper describes the implementation of a target independent LLVM pass that is executed before vectorization, and which replaces larger integer data types with smaller ones for targets that have support for arithmetic using multiple integer sizes, and in particular for vector arithmetic support with several integer sizes; while also ensuring that the correctness of the results is retained.

The Movidius SHAVE processor has native support for vectors of 8-bit, 16-bit and 32-bit integers, so reversing the widening that occurs due to UACs presents greater opportunities for optimization using higher ordinal vectors of smaller integers.

Introduction

While implementing the vectorizing optimizations for the Movidius SHAVE processor, we noticed that the compiler was not selecting the most optimal vector instructions for the problem as expressed in C, and decided to investigate why this was the case.

What we found was that there was a contention between the goals of the front-end to honor and respect the ISO C and ISO C++ rules for arithmetic, and the goals of the back-end to select the best instructions for the task. In particular, we discovered that the "Usual Arithmetic Conversions" [UACs] were at odds with the vectorizer.

The Problem

The SHAVE processor like many processors which provide SIMD instructions, has optimized instructions for handling vectors of 8-bit, 16-bit and 32-bit integers. In our architecture we natively support 128-bit and 32-bit SIMD vectors. The 128-bit SIMD instructions operate on the following LLVM integer vector types:

`v4i32`, `v8i16` and `v16i8`

While the 32-bit SIMD instructions operate on:

`v2i16` and `v4i8`

I'll use the following abbreviated example to illustrate the problem:

```
const unsigned char * __restrict__ src1;
const unsigned char * __restrict__ src2;
unsigned char * __restrict__ dst;
int index;

for (index = 0; index < 256; ++index)
  dst[index] = src1[index] + src2[index];
```

What we expected to see was the vectorizer optimizing this to use '`v16i8`' instructions. Instead the vectorizer was unable to optimize it at all and the generated code used scalar '`i8`' instructions. It was only after we had thoroughly checked that the pattern matching in the vectorizer was correct with '`.ll`' tests, that we had a look at the output from the front-end, and the IR after each pass.

Initially we were surprised to see that the vectorizer was unable to optimize this example. Then we had the "duh!" moment when we realized that the problem was with the expression:

```
src1[index] + src2[index]
```

ISO C and C++ rules specifically state that this expression is evaluated using the '`int`' data type, which in our case is 32-bits. So the front-end correctly widens the two source operands, performs the arithmetic, and narrows the result. It was this widening of the operands and subsequent narrowing of the result that prevented vectorization from taking place.

But we know of course that an integer ADD like this using modulo-256 arithmetic will yield the same binary result, and that the 32-bit evaluation is unnecessary; we are also not concerned with integer overflow and underflow.

Our Resolution

What we did to resolve this was to introduce a new pass, as we couldn't find one that already did this. This new pass we called the "`ExtendTruncateReduction`" pass, though we also commonly refer to it as the UAC reduction pass.

This is a relatively straightforward pass that makes good use of the ISO Standards "as-if" rule. It replaces chains of instructions that begin with "extend" instructions and end with "truncate" instructions with equivalent chains which use a "smaller" data-type.

Take the following sequence of LLVM-IR instructions for example:

```
%0 = load i16, i16* @a, align 2, !tbaa !1
%conv.3 = zext i16 %0 to i32
%1 = load i16, i16* @b, align 2, !tbaa !1
%conv1.4 = zext i16 %1 to i32
%add = add nuw nsw i32 %conv1.4, %conv.3
%conv2 = trunc i32 %add to i16
store i16 %conv2, i16* @c, align 2, !tbaa !1
```

The UAC reduction pass replaces this chain of instructions with:

```
%0 = load i16, i16* @a, align 2, !tbaa !1
%1 = load i16, i16* @b, align 2, !tbaa !1
%2 = add i16 %1, %0
store i16 %2, i16* @c, align 2, !tbaa !1
```

The process by which the pass performs this transformation can be broadly described with the following four steps:

1. Find a truncate instruction in the current function.
2. Generate a list of instructions that need to be modified in order to remove this truncate instruction.
3. Check if it is "safe" to perform the transformation on the generated list.
4. If it is "safe" then perform the transformation.

The list of instructions that need to be modified is generated iteratively, starting with the initial truncate instruction. All uses of an instruction are added to the list, as well as all instructions that define an operand of that instruction. Once an instruction has been added to the list, all of its own uses and operands are checked in the same manner. The operands of "extend" instructions are not checked, neither are the uses of "truncate" instructions.

The following set of conditions have to be met for a list of instructions to be considered "safe" to modify:

1. All entries in the list must be a constant, "truncate", "extend", or "BinaryOperator" instruction.
2. All instructions must have at least one use.
3. An instruction cannot be a "divide" or "remainder" unless all operands are defined by an "extend" instruction or are constant.
4. An instruction cannot be a shift right unless its LHS operand is defined by a shift left instruction, where both RHS operands are a constant value 32 minus the bit size of the type being converted to.

Assuming the above criteria have been met for all instructions in the list, the transformation is then performed. This is done simply by maintaining a map of "original" instructions to "new" instructions. If all instructions defining operands of an instruction have their own corresponding "new" instruction, that instruction can be transformed as well. Once all instructions in the list have been transformed in this way, the actual substitution is performed. Uses of extend instructions are replaced by the value being extended. Uses of truncate instructions are replaced by the "new" value that was originally truncated.

Since this optimization is inserted before vectorization, the information the vectorizer sees uses the smallest valid integer types and the back-end can now select the most appropriate SIMD instructions.

Interactions and Future Directions

There are some interactions with the target machine, but only in the high-level sense. For instance, there is no point in applying this reduction pass if the target machine

cannot perform integer arithmetic in multiple integer sizes, as it would just have to undo the effect of this pass anyway. So the pass can be selectively enabled for targets that do have such arithmetic support, and it also performs checks on the target machine to see if it is worth while performing the transformation.

Although the pass itself is target independent, there are some minor tuning issues regarding loop-unrolling that we are examining to see if they are specific to our back-end, or to see if we need to extend the interface to the "`TargetTransformInfo`" class so that targets can implement their own hooks for tuning this pass.

When these issues are resolved, we hope to propose the inclusion of this pass into the mainline sources as we believe that it is a useful target-independent pass that will also benefit targets other than the SHAVE.

Performance Impact

Applying this pass prior to vectorization has resulted in considerable performance improvements, not only because the vectorizer is able to select more optimal arithmetic instructions, but the reduced cost of eliminating the unnecessary extending and truncating instructions has further contributed to the performance gains.

For our testing we compared the results from 1,849 tests. Not surprisingly, the majority of tests that saw a performance improvement are in the TSVC group of tests, and others in both the GCC vector tests and in LibC++ containers and iterators.

There were regressions too, but in each case where a regression occurred, the problem was easily corrected by adding:

```
#pragma unroll(X)
```

To the appropriate loop, where X is some integer constant (this value changed from test to test). In one case, adding '`inline`' to the declaration of a key function that was no longer being inlined. Both of these are likely to be due to poor tuning in the thresholds we have set in our "`TargetTransformInfo`" specialization. We are still looking into this.

The breakdown of these tests 1,849 is as follows:

- 2 tests are more than 2X slower
- 7 are more than 10% slower
- 35 are more than 5% slower
- 52 are more than 5% faster
- 82 are more than 10% faster
- 22 are more than 2X faster
- 17 are more than 5X faster
- 2 are more than 10X faster
- 1 is more than 20X faster
- 2 are more than 30X faster

The original work was done on the LLVM v3.6 sources, but the above figures are from the LLVM v3.7 version.