LLVM SIMD-to-SWAR Pass
Shan Cao
Christian Millar

The goal of this project is to modify LLVM to handle arbitrary vectors. Vectors in LLVM are used to indicate SIMD usage - a vector of type <8x2i> contains eight items, each of length 2, for a total vector length of 16 bytes. However, while LLVM's grammar allows arbitrary vectors, implementations rarely support them.
Vectors are implemented by LLVM backends as direct uses of the processor's SIMD registers. The SIMD registers are of fixed length, and have variable amounts of partitioning available. This means that common vectors (containing eg i4's, i8's, i16's) are well supported. However, if you have a vector containing, for example <12x3i>, for 36 bytes of storage, that is unlikely to fit in a SIMD register, and the partitioning offered by the register will be ill-suited to your 3-byte values.
Our goal is to automatically convert your <12x3i> vector and operations using it into similarly efficient code using registers to store your values, and modified operations that preserve the speed of SIMD.

Several types of operations are available for SWAR:

Polymorphic operations allow us to use the same operation, unmodified. Bitwise AND is a good example of this. Vector form:
|001101| |101111| & |110100| |111000| = |000100| |101000|
yields the same result as combining our entries in a register:
001101101111 & 110100111000 = 000100101000

Partitioned operations require modified operations; for them, each 'word' we have in the register has our operation performed on it, and is not allowed to interfere with other words. Addition is a good example: we would not want overflow from our first word to affect the sum stored in the second word. We will need to generate LLVM to guarantee that this is the case when modifying these operations.

Communication operations allow values to read from other fields in the vector. For instance, when running a blurring algorithm on an image, you would need to read the values of adjacent pixels in order to compute the value of a single field.

Reductions combine elements in a SIMD vector, usually according to some tree-based reduction. A similar procedure can be applied to data in registers, again accounting for overflow.

Masking operations are common in SIMD, and more so in SWAR. These operations can be performed with logical operations with constants to clear fields, as opposed to the hardware multiplexers often found in SIMD implementations.

For an example of a partitioned operation, we can look to this example:

```
@firstvec = global <6 x i3> <i3 1, i3 2, i3 3, i3 3, i3 2, i3 2>
@secondvec = global <6 x i3> <i3 3, i3 2, i3 1, i3 1, i3 2, i3 3>

define i32 @main(){
      %a1 = load <6 x i3>, <6 x i3>* @firstvec
      %a2 = load <6 x i3>, <6 x i3>* @secondvec
      %r = add <6 x i3> %a1, %a2
      ret i32 0
}
```

The vectors here are not going to fit in a standard SIMD register. However, we can convert them before the operation into a single integer, then expand the addition to handle overflow:

```
@firstvec = global <6 x i3> <i3 1, i3 2, i3 3, i3 3, i3 2, i3 2>
@secondvec = global <6 x i3> <i3 3, i3 2, i3 1, i3 1, i3 2, i3 3>

define i32 @main(){
      %v1 = load <6 x i3>, <6 x i3>* @firstvec
      %v2 = load <6 x i3>, <6 x i3>* @secondvec

      %a1 = bitcast <6 x i3> %v1 to i18
      %a2 = bitcast <6 x i3> %v2 to i18

      %m1 = and i18 %a1, 112347 ;constants used to mask out high bits
      %m2 = and i18 %a2, 112347
      %r1 = add i18 %m1, %m2
      %r2 = xor i18 %m1, %m2
      %r3 = and i18 %r2, 149796 ;constants used to check high bit values
      %r4 = xor i18 %r1, %r3

      %r = bitcast i18 %r4 to <6 x i3>
      ret i32 0
}
```

Here we do somewhat more work; first we perform the cast, which gives us integers. Then we AND those integers with a constant, 112347. This constant is used because in binary, it is of the form 011 011 011 011 011 011, so AND-ing our integers will mask out the high bits of each field. This gives us space to perform addition, without worrying about fields overlapping due to overflow (this sum is stored in %r1). Then, all that remains is to handle the most significant bit of each field. We XOR the integers, which gives each most significant bit the effect of being added without accounting for overflow. Then we AND that result with another constant, 149796 (binary: 100 100 100 100 100 100), which masks out everything but the most significant bits. Finally, we XOR our most significant bits with the less significant bits, yielding an accurate integer, which can be cast back to a vector.

It is worth noting that this is rather more work than a simple addition. However, speed gains can still occur with this technique; faced with vectors that do not fit in SIMD registers, LLVM can choose to perform this addition sequentially, yielding O(n) performance. By way of contrast, this technique will work for arbitrary vectors, and exploit parallelism to get much higher performance.

Citations

http://www.tldp.org/HOWTO/Parallel-Processing-HOWTO-4.html
http://www.phys.aoyama.ac.jp/~w3-furu/docs/aoyama+/Tech_notes/adaptor_doc/Users_Guide.pdf