Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 02 May 2017 21:55:53 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 02 May 2017 22:54Hi Mirek,

I realized that my pipelined function was wrong. I made a silly mistake in the test case. The function is now fixed as:

```
// Testing pipelined version of Bits::Set(int, bool)
void Bits::PipelineSet(int i, const dword bs)
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Do we need to expand the internal buffer first?
 // Also check whether we can place 4 bits in the existing DWORD. If not, we should expand.
 if(q >= alloc)
  Expand(q);

 // Get integer bit values according to existing DWORD value and indices.
 // Assuming default value of bool is 0x1 if true!
 const dword d1 = !!(bs & PowersOfTwo[0]) << i;
 const dword d2 = !!(bs & PowersOfTwo[8]) << (i + 1);
 const dword d3 = !!(bs & PowersOfTwo[16]) << (i + 2);
 const dword d4 = !!(bs & PowersOfTwo[24]) << (i + 3);
 bp[q] = (bp[q] | d1 | d2 | d3 | d4);
}
```

I did some more tests, and I am really surprised by the amount of difference the compiler and CPU make in this situation. I had to switch from my i7 2600k CPU to an Intel Core i7 4710MQ CPU because I was missing AVX2 (AVX2 really made it fast) :) When compiled with the Visual C++ compiler, I got the following result.

I was surprised to see how the newer CPU runs the simple pipelined version a lot faster than the 2600k! I also saw that a vectorized version of the Set function is almost simpler than the regular one. :) However, when I use the Intel C++ compiler, the results are very different:

It seems that the Intel compiler generates way different code, making the SSE2 version blazingly fast. 10 times faster than the regular Set function. The strange thing is, that when I use the Visual C++ compiler, the timing of the different test cases is as I expected them to be. I expected the AVX2 function to be faster than the SSE2 one, which clearly is not the case with the Intel compiler.

The sources of my vectorized functions is as following:

```
// Testing vectorized version of Bits::Set(int, bool)
// We require that the input bools have value 0x80, e.g. most significant byte set if true.
void Bits::VectorSet(int i, const unsigned char vec[16])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m128i boolVec = _mm_set_epi8(vec[15], vec[14], vec[13], vec[12], vec[11], vec[10], vec[9],
vec[8]
  , vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm_movemask_epi8(boolVec);

 // Set the resulting WORD.
 LowHighDword w;
 w.dw = bp[q];
 if (i == 16)
 {
  w.w2 = (short)bitMask;
 }
 else
 {
  w.w1 = (short)bitMask;
 }
 bp[q] = w.dw;
}
```

```
// The same vectorized function, but with AVX2 instructions.
void Bits::VectorSetAVX2(int i, const unsigned char vec[32])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m256i boolVec = _mm256_set_epi8(vec[31], vec[30], vec[29], vec[28], vec[27], vec[26],
vec[25], vec[24], vec[23]
  , vec[22], vec[21], vec[20], vec[19], vec[18], vec[17], vec[16], vec[15], vec[14], vec[13], vec[12],
vec[11]
  , vec[10], vec[9], vec[8], vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm256_movemask_epi8(boolVec);

 // Set the resulting DWORD.
 bp[q] = bitMask;
}
```

Is it feasible to make a vectorized version for U++ by default, or should I provide it for myself?

Thanks,

evo

I still think there is a glitch somewhere benchmarking this. Would it possible to post a whole package here?

Mirek