Counting array elements which are below a particular limit value using SSE

devblogs.microsoft.com/oldnewthing/20141201-00

December 1, 2014

Raymond Chen

Some time ago, we looked at how doing something can be faster than not doing it. That is, we observed the non-classical effect of the branch predictor. I took the branch out of the inner loop, but let's see how much further I can push it.

The trick I'll employ today is using SIMD in order to operate on multiple pieces of data simultaneously. Take the original program and replace the `countthem` function with this one:

```
int countthem(int boundary)
{
 __m128i xboundary = _mm_cvtsi32_si128(boundary);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000; i++) {
  __m128i value =  _mm_cvtsi32_si128(array[i]);
  __m128i test = _mm_cmplt_epi32(value, xboundary);
  count = _mm_sub_epi32(count, test);
 }
 return _mm_cvtsi128_si32(count);
}
```

Now, this program doesn't actually use any parallel operations, but it's our starting point. For each 32-bit value, we load it, compare it agains the boundary value, and accumulate the result. The `_mm_cmplt_epi32` function compares the four 32-bit integers in the first parameter against the four 32-bit integers in the second parameter, producing four new 32-bit integers. Each of the new 32-bit integers is `0xFFFFFFFF` if the corresponding first parameter is less than the second, or it is `0x00000000` if it is greater than or equal.

In this case, we loaded up the value we care about, then compare it against the boundary value. The result of the comparison is either 32 bits of 0 (for false) or 32 bits of 1 (for true), so this merely sets `test` equal to `0xFFFFFFFF` if the value is less than the boundary; otherwise `0x0000000`. Since `0xFFFFFFFF` is the same as a 32-bit `-1`, we subtract the value so that the count goes up by 1 if the value is less than the boundary.

Finally, we convert back to a 32-bit integer and return it.

With this change, the running time drops from 2938 time units to 2709, an improvement of 8%.

So far, we have been using only the bottom 32 bits of the 128-bit XMM registers. Let's turn on the parallelism.

```
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary = _mm_set1_epi32(boundary);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000 / 4; i++) {
  __m128i value = _mm_loadu_si128(&xarray[i]);
  __m128i test = _mm_cmplt_epi32(value, xboundary);
  count = _mm_sub_epi32(count, test);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return _mm_cvtsi128_si32(count);
}
```

We take our 32-bit integers and put them in groups of four, so instead of thinking of them as 10000 32-bit integers, we think of them as 2500 128-bit blocks, each block containing four *lanes*, with each lane holding one 32-bit integers.

	Lane 3	Lane 2	Lane 1	Lane 0
`xarray[0]`	`array[3]`	`array[2]`	`array[1]`	`array[0]`
`xarray[1]`	`array[7]`	`array[6]`	`array[5]`	`array[4]`
⋮	⋮	⋮	⋮	⋮
`xarray[2499]`	`array[9999]`	`array[9998]`	`array[9997]`	`array[9996]`

Now we can run our previous algorithm in parallel on each lane.

	Lane 3	Lane 2	Lane 1	Lane 0
`xboundary`	`boundary`	`boundary`	`boundary`	`boundary`

`test`	`array[3] < boundary`	`array[2] < boundary`	`array[1] < boundary`	`array[0] < boundary`
`test`	`array[7] < boundary`	`array[6] < boundary`	`array[5] < boundary`	`array[4] < boundary`
⋮	⋮	⋮	⋮	⋮
`test`	`array[9999] < boundary`	`array[9998] < boundary`	`array[9997] < boundary`	`array[9996] < boundary`
`count` = Σ − `test`	Lane 3 totals	Lane 2 totals	Lane 1 totals	Lane 0 totals

The `xboundary` variable contains a copy of the boundary in each of the four 32-bit lanes. We load the values from the array four at a time[1] and compare them (in parallel) against the boundary, then we tally them (in parallel). The result of the loop is that each lane of `count` performs a count of values for its lane.

After we complete the loop, we combine the parallel results by adding the lanes together. We do this by shuffling the values around and performing more parallel adds. The `_mm_shuffle_epi32` function lets you rearrange the lanes of an XMM register. The `_MM_SHUFFLE` macro lets you specify how you want the shuffle to occur. For example, `_MM_SHUFFLE(1, 0, 3, 2)` says that we want lanes 1, 0, 3 then 2 of the original value. (You can shuffle a value into multiple destination lanes; for example, `_MM_SHUFFLE(0, 0, 0, 0)` says that you want four copies of lane 0. That's how we created `xboundary`.)

	Lane 3	Lane 2	Lane 1	Lane 0
`count`	Lane 3 totals	Lane 2 totals	Lane 1 totals	Lane 0 totals
`shuffle1`	Lane 1 totals	Lane 0 totals	Lane 3 totals	Lane 2 totals
`count += shuffle1`	Lane 3 + Lane 1	Lane 2 + Lane 0	Lane 1 + Lane 3	Lane 0 + Lane 2
`shuffle2`	Lane 2 + Lane 0	Lane 3 + Lane 1	Lane 0 + Lane 2	Lane 1 + Lane 3

`count += shuffle2`	Lane 3 + Lane 1 + Lane 2 + Lane 0	Lane 2 + Lane 0 + Lane 3 + Lane 1	Lane 1 + Lane 3 + Lane 0 + Lane 2	Lane 0 + Lane 2 + Lane 1 + Lane 3

At the end of the shuffling and adding, we have calculated the sum of all four lanes. (For style points, I put the answer in all the lanes.)

This new version runs in 688 time units, or 3.9 times faster than the previous one. This makes sense because we are counting four values at each iteration. The overall improvement is 4.3×.

Let's see if we can reduce the loop overhead by doing some unrolling.

```
#define GETVALUE(n) __m128i value##n = _mm_loadu_si128(&xarray[i+n])
#define GETTEST(n) __m128i test##n = _mm_cmplt_epi32(value##n, xboundary)
#define GETCOUNT(n)  count = _mm_sub_epi32(count, test##n)
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary = _mm_set1_epi32(boundary);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000 / 4; i += 4) {
  GETVALUE(0); GETVALUE(1); GETVALUE(2); GETVALUE(3);
   GETTEST(0);  GETTEST(1);  GETTEST(2);  GETTEST(3);
  GETCOUNT(0); GETCOUNT(1); GETCOUNT(2); GETCOUNT(3);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return _mm_cvtsi128_si32(count);
}
```

We unroll the loop fourfold. At each iteration, we load 16 values from memory, and then accumulate the totals. We fetch all the memory values first, then do the comparisons, then accumulate the results. If we had written it as `GETVALUE` immediately followed by `GETTEST`, then the `_mm_cmplt_epi32` would have stalled waiting for the result to arrive from memory. By interleaving the operations, we get some work done instead of stalling.

This version runs in 514 time units, an improvement of 33% over the previous version and an overall improvement of 5.7×.

Can we unroll even further? Let's try fivefold.

```
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary = _mm_set1_epi32(boundary);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000 / 4; i += 5) {
  GETVALUE(0); GETVALUE(1); GETVALUE(2); GETVALUE(3); GETVALUE(4);
   GETTEST(0);  GETTEST(1);  GETTEST(2);  GETTEST(3);  GETTEST(4);
  GETCOUNT(0); GETCOUNT(1); GETCOUNT(2); GETCOUNT(3); GETCOUNT(4);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return _mm_cvtsi128_si32(count);
}
```

Huh? This version runs marginally *slower*, at 528 time units. So I guess further unrolling won't help any more. (For example, if you unroll a loop so much that you have more live variables than registers, the compiler will need to spill registers to memory. The x86 has eight XMM registers available, so you can easily cross that limit.)

But wait, there's still room for tweaking. We have been using `_mm_cmplt_epi32` to perform the comparison, expecting the compiler to generate code like this:

```
    ; suppose xboundary is in xmm0 and count is in xmm1
    movdqu   xmm2, xarray[i] ; xmm2 = value
    pcmpltd  xmm2, xmm0      ; xmm2 = test
    psubd    xmm1, xmm2
```

If you crack open your Intel manual, you'll see that there is no `PCMPLTD` instruction. The compiler intrinsic is emulating the instruction by flipping the parameters and using `PCMPGTD`.

```
_mm_cmplt_epi32(x, y) ↔ _mm_cmpgt_epi32(y, x)
```

But the `PCMPGTD` instruction writes the result back into the first parameter. In other words, it always takes the form

```
y = _mm_cmpgt_epi32(y, x);
```

In our case, `y` is `xboundary`, but we don't want to modify `xboundary`. As a result, the compiler needs to introduce a temporary register:

```
    movdqu   xmm2, xarray[i] ; xmm2 = value
    movdqa   xmm3, xmm0      ; xmm3 = copy of xboundary
    pcmpgtd  xmm3, xmm2      ; xmm3 = test
    psubd    xmm1, xmm3
```

We can take an instruction out of the sequence by switching to `_mm_cmpgt_epi32` and adjusting our logic accordingly, taking advantage of the fact that

```
x < y ⇔ ¬(x ≥ y) ⇔ ¬(x > y - 1)
```

assuming the subtraction does not underflow. Fortunately, it doesn't in our case since `boundary` ranges from 0 to 10, and subtracting 1 does not put us in any danger of integer underflow.

With this rewrite, we can switch to using `_mm_cmpgt_epi32`, which is more efficient for our particular scenario. Since we are now counting the values which *don't* meet our criteria, we need to take our final result and subtract it from 10000.

```
#define GETTEST(n) __m128i test##n = _mm_cmpgt_epi32(value##n, xboundary1)
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary1 = _mm_set1_epi32(boundary - 1);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000 / 4; i += 5) {
  GETVALUE(0); GETVALUE(1); GETVALUE(2); GETVALUE(3); GETVALUE(4);
   GETTEST(0);  GETTEST(1);  GETTEST(2);  GETTEST(3);  GETTEST(4);
  GETCOUNT(0); GETCOUNT(1); GETCOUNT(2); GETCOUNT(3); GETCOUNT(4);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return 10000 - _mm_cvtsi128_si32(count);
}
```

Notice that we have two subtractions which cancel out. We are subtracting the result of the comparison, and then we subtract the total from 10000. The two signs cancel out, and we can use addition for both. This saves an instruction in the `return` because subtraction is not commutative, but addition is.

```
#define GETCOUNT(n) count = _mm_add_epi32(count, test##n)
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary1 = _mm_set1_epi32(boundary - 1);
 __m128i count = _mm_setzero_si128();
 for (int i = 0; i < 10000 / 4; i += 5) {
  GETVALUE(0); GETVALUE(1); GETVALUE(2); GETVALUE(3); GETVALUE(4);
   GETTEST(0);  GETTEST(1);  GETTEST(2);  GETTEST(3);  GETTEST(4);
  GETCOUNT(0); GETCOUNT(1); GETCOUNT(2); GETCOUNT(3); GETCOUNT(4);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return 10000 + _mm_cvtsi128_si32(count);
}
```

You can look at the transformation this way: The old code considered the glass half empty. It started with zero and added 1 each time it found an entry that passed the test. The new code considers the glass half full. It assumes each entry passes the test, and it subtracts one each time it finds an element that fails the test.

This version runs in 453 time units, an improvement of 13% over the fourfold unrolled version and an improvement of 6.5× overall.

Okay, let's unroll sixfold, just for fun.

```
int countthem(int boundary)
{
 __m128i *xarray = (__m128i*)array;
 __m128i xboundary = _mm_set1_epi32(boundary - 1);
 __m128i count = _mm_setzero_si128();
 int i = 0;
 {
    GETVALUE(0); GETVALUE(1); GETVALUE(2); GETVALUE(3);
     GETTEST(0);  GETTEST(1);  GETTEST(2);  GETTEST(3);
    GETCOUNT(0); GETCOUNT(1); GETCOUNT(2); GETCOUNT(3);
 }
 i += 4;
 for (; i < 10000 / 4; i += 6) {
  GETVALUE(0); GETVALUE(1); GETVALUE(2);
  GETVALUE(3); GETVALUE(4); GETVALUE(5);
   GETTEST(0);  GETTEST(1);  GETTEST(2);
   GETTEST(3);  GETTEST(4);  GETTEST(5);
  GETCOUNT(0); GETCOUNT(1); GETCOUNT(2);
  GETCOUNT(3); GETCOUNT(4); GETCOUNT(5);
 }
 __m128i shuffle1 = _mm_shuffle_epi32(count, _MM_SHUFFLE(1, 0, 3, 2));
 count = _mm_add_epi32(count, shuffle1);
 __m128i shuffle2 = _mm_shuffle_epi32(count, _MM_SHUFFLE(2, 3, 0, 1));
 count = _mm_add_epi32(count, shuffle2);
 return 10000 + _mm_cvtsi128_si32(count);
}
```

Since `10000 / 4 % 6 = 4`, we have four values that don't fit in the loop. We deal with those values up front, and then enter the loop to get the rest.

This version runs in 467 time units, which is 3% slower than the previous version. So I guess it's time to stop unrolling. Let's go back to the previous version which ran faster.

The total improvement we got after all this tweaking is speedup of 6.5× over the original jumpless version. And most of that improvement (5.7×) came from unrolling the loop fourfold.

Anyway, no real moral of the story today. I just felt like tinkering.

Notes

[1] The `_mm_loadu_si128` intrinsic is kind of weird. Its formal argument is a `__m128i*`, but since it is for loading unaligned data, the formal argument really should be `__m128i __unaligned*`. The problem is that the `__unaligned` keyword doesn't exist on x86 because prior to the introduction of MMX and SSE, x86 allowed arbitrary misaligned data. Therefore, you are in this weird situation where you have to use an aligned pointer to access unaligned data.

Bonus chatter: Clang at optimization level 3 does autovectorization. It doesn't know some of the other tricks, like converting `x + 1` to `x - (-1)`, thereby saving an instruction and a register.

Raymond Chen

Follow