Let Me Count The Ways: Optimizing 64 Choose 4

Paul D. Senzee
Software Engineer
EA Tiburon
psenzee@ea.com

I pulled this challenge from a conversation Ryan Burkett and I had a few months ago. I'd been tinkering with card games a bit and one of the little questions that came up was *what is the fastest way to enumerate all possible combinations of 4 items out of a possible 64?* Actually, it doesn't have to be 4 of 64, it could be 7 of 48, or 2 specific aces from 52 cards or 3 bits of 10. This is expressed as the binomial coefficient { n choose r } and has the formula *factorial(n) / (factorial(r) * factorial(n - r))*. In the case of *n = 64* and *r = 4* it yields 635,376 different combinations.

This became Hacker's Delight #6 – 64 Choose 4. The task is to enumerate each of these unique combinations exactly one time until all 635,376 have been generated.

An off-the-cuff decision we made when putting the problem together drastically affected the character of this problem. The question was *should the problem require filling a buffer or simply returning every possible combination in sequence*. Filling a buffer seemed simpler. Unexpectedly, the problem transformed from an algorithmic style problem to an unimaginative and hardware-dependent "how fast can you fill the buffer" problem. While I'm not entirely happy with that, it has been educational.

The Naive Approach

The most basic brute force approach is to iterate through every value that can be contained in 64 bits and check if it has four bits. How long would this take? On my machine it takes 1 second to check 250 million numbers. At this rate it will take 2,340 years to check all 2^{64} numbers!

A Better Approach

Poking around I found a function that takes a number and returns the next highest number with the same number of bits. I adapt it a bit and bam!

```
enum { CHOOSE_64_4 = 635376 };

void Choose64_4_Senzee_slow(unsigned __int64 *v)
{
    unsigned __int64 y, q, x;
    v[0] = ((unsigned __int64)1 << 4) - 1;
    for (unsigned i = 1; i < CHOOSE_64_4; i++)
    {
        x    = v[i - 1];
        y    = x & -(__int64)x;
        q    = x + y;
        v[i] = q | (((x ^ q) >> 2) / y);
    }
}
```

This variation turns in 32 milliseconds (~180 cycles per element) for complete enumeration. Not a bad improvement over 2,340 years, eh?

An Even Better Approach

At first glance it seemed that the above would blow this one away. Not so. One problem with the above approach is that it has a nasty little division. That's a 64-bit division and for reasons explained below regarding 64-bit shifts, 64-bit math in C++ as compiled by Visual C++ is just not very optimal. Another problem with the above function is that there's just way too much going on in that inner loop. After all, it is being called over half a million times. Ideally the inner loop would be super tight.

```
////////////////////////////////////////////
// Recursive implementation

#include <vector>

void enumerate(int n, int r, std::vector<unsigned __int64> &v, unsigned __int64 hi = 0)
{
    if (r != 0)
    {
        if (r == 1)
        {
            unsigned __int64 mask = 1;
            for (int i = 0; i < n; i++)
            {
                v.push_back(mask | hi);
                mask <<= 1;
            }
        }
        else if (r == n)
        {
            v.push_back(((1ui64 << n) - 1) | hi);
        }
        else
        {
            enumerate(n - 1, r,     v,                      hi);
            enumerate(n - 1, r - 1, v, (1ui64 << (n - 1)) | hi);
        }
    }
}
```

The next is based on the initial recursive approach above, but the recursion and std::vector removed. In place of the recursion an explicit stack is used (`s[64]`). The std::vector allocates and frees memory, and that's just not appropriate for highly optimized code. Note that for the last enumerate call, we've got a case of tail recursion. That can be replaced without pushing or popping the stack - just a goto. The following `Choose64_4_Senzee_basic` runs to completion in just 3.9 milliseconds (at ~22 cycles per element), almost 10 times faster than `Choose64_4_Senzee_slow`.

```
////////////////////////////////////////////
// Core algorithm

void Choose64_4_Senzee_basic(unsigned __int64 *v)
{
    struct { int n, r; unsigned __int64 h; } s[64] = { { 64, 4, 0 } }, q;
    int si = 1;
    while (si)
    {
        q = s[--si];                         // executed 39711 times
tail:
        if (q.r != 0)
        {
            if (q.r == 1)
            {                                // executed 37820 times
                for (int j = 0; j < q.n; j++)
                    *v++ = q.h | (1ui64 << j);   // executed 633485 times
            }
            else if (q.r == q.n)
            {
                *v++ = q.h | (1ui64 << q.n) - 1; // executed 1891 times
            }
            else
            {
                --q.n; s[si++] = q; q.r--;       // executed 39710 times
                q.h |= 1ui64 << q.n;
                goto tail;
            }
        }
    }
}
```

As it turns out, 64 bit shifts are killer as compiled by Visual C++. For some reason, the designers of the compiler opted not to use the 64-bit shift instructions provided by the MMX extensions (even when turned on in the compile options) and instead the compiler actually generates and calls an *uninlined* function called _all_shl. The function even has a branch in it! Replacing `(1ui64 << x)` with the following table chopped about *1.4ms* off of the running time of the algorithm.

```
///////////////////////////////////////////////////
// Replaced shifts with table
// My fastest C++ version at about 15.3 cycles per element and ~2.5ms

namespace
{
const unsigned __int64 _one_shift[] =
{
    1ui64 <<  0, 1ui64 <<  1, 1ui64 <<  2, 1ui64 <<  3,
    1ui64 <<  4, 1ui64 <<  5, 1ui64 <<  6, 1ui64 <<  7,
    1ui64 <<  8, 1ui64 <<  9, 1ui64 << 10, 1ui64 << 11,
    1ui64 << 12, 1ui64 << 13, 1ui64 << 14, 1ui64 << 15,
    1ui64 << 16, 1ui64 << 17, 1ui64 << 18, 1ui64 << 19,
    1ui64 << 20, 1ui64 << 21, 1ui64 << 22, 1ui64 << 23,
    1ui64 << 24, 1ui64 << 25, 1ui64 << 26, 1ui64 << 27,
    1ui64 << 28, 1ui64 << 29, 1ui64 << 30, 1ui64 << 31,
    1ui64 << 32, 1ui64 << 33, 1ui64 << 34, 1ui64 << 35,
    1ui64 << 36, 1ui64 << 37, 1ui64 << 38, 1ui64 << 39,
    1ui64 << 40, 1ui64 << 41, 1ui64 << 42, 1ui64 << 43,
    1ui64 << 44, 1ui64 << 45, 1ui64 << 46, 1ui64 << 47,
    1ui64 << 48, 1ui64 << 49, 1ui64 << 50, 1ui64 << 51,
    1ui64 << 52, 1ui64 << 53, 1ui64 << 54, 1ui64 << 55,
    1ui64 << 56, 1ui64 << 57, 1ui64 << 58, 1ui64 << 59,
    1ui64 << 60, 1ui64 << 61, 1ui64 << 62, 1ui64 << 63
};
}

void Choose64_4_Senzee(unsigned __int64 *v)
{
    typedef unsigned __int64 ui64;
    struct { int n, r; ui64 h; } s[64] = { { 64, 4, 0 } }, q;
    int si = 1;
    while (si)
    {
        q = s[--si];
tail:
        if (q.r != 0)
        {
            if (q.r == 1)
            {
                for (int j = 0; j < q.n; j++)
                    *v++ = q.h | _one_shift[j];
            }
            else if (q.r == q.n)
            {
                *v++ = q.h | _one_shift[q.n] - 1;
            }
            else
            {
                --q.n; s[si++] = q; q.r--;
                q.h |= _one_shift[q.n];
                goto tail;
            }
        }
    }
}
```

Shortly, solutions came in that were almost exactly as fast (~2.5ms ~15 cycles per element) with quite different algorithms. Hmm, I thought - *the bottleneck must be memory-processor bandwidth - ~2.5ms is simply how long it takes to write 4.8mb of data. All the other processing is swamped by that.* This seemed to be confirmed by the fact that ~2.5ms is actually (very slightly) faster than the pure memset “solution” below!

```
////////////////////////////////////////////////////
// memset takes ~2.5ms to fills the 4.8mb buffer with 0x44
// This does not generate the correct answer!

enum { CHOOSE_64_4 = 635376 };

void Choose64_4_Senzee_memset_reference(unsigned __int64 *pDest)
{
    memset(pDest, 0x44, sizeof(unsigned __int64) * CHOOSE_64_4); // wrong answer, but
                                                                  // we want to time it.
}
```

That conviction was undermined when Jim Hejl came up with an SSE version of "memset" that executed in about half memset's time. Then a wicked entry came from Graham Hazel at EA United Kingdom that executed in ~1.25ms (~7.5 cycles per element) - all SIMD (MMX)! Knowing now that it was possible and asking that it not be forwarded to me, I became obsessed.

A Change in Character - Moving to Assembly

Our problem had now become the unimaginative buffer filling problem. Initially, I tried assembly only for the inner loops. Interestingly, putting a single __asm nop; into the code seemed to cripple the VC++ optimizer. It appears the optimizer gives up trying to optimize blocks of code (perhaps the entire function) once you insert an __asm block. The whole function had to go __asm. Even so, the first full assembly versions I came up with were slower than the C++ version above.

Cache Management

Like in Hacker's Delight #4: Counting Primes the key to getting an optimal solution is good cache management. Jim's SSE memset used the movnt* (move-non-temporal) instructions to bypass the cache when writing the results into the buffer. The same is done here. The recent Intel chips have a decoupled memory architecture that enables instructions to be executed while a memory access is taking place asynchronously. This can only happen, of course, if there are no data dependencies with the following instructions. Using movnt* indicates to the processor that we are not interested in reading back the value so it can be written out-of-order and that we do not want to load the cache with data at that address. Between those writes, we can hide the rest of the inner loop's cycles. This should give us near saturation of the memory bus.

We want the stack in the cache, however. It's tiny with a maximum depth of three (r – 1 from {n choose r}?). This keeps the cache sparkling clean with just the stack and nothing else. The stack being so small inspired a version where the stack is shoved into xmm (SSE) registers with each 32-bit piece of data split among them. Pushing and popping was done with left and right shifting, respectively, of these registers. There was a fair amount of overhead associated with this approach. In the end it was slower than the in-cache stack version.

The first good pass at MMX version came in about 8.2 – 8.6 cycles per element.

```
/// first fast mmx version

// mmx version of Choose64_4_Senzee()
void Choose64_4_Senzee_mmx(unsigned __int64 *v)
{
    __declspec(align(16))
    struct { int n, r; unsigned __int64 h; } s[64] = { { 64, 4, 0 } };
    void *ps = &s[0];
    __asm
    {
        // registers
        // ---------
        // eax = q.n        ebx = <don't use>
        // ecx = scratch    edx = q.r
        // edi = ps         esi = v
        // mm7 = 1          mm6 = q.h
        // mm5 = ps [base]

        mov eax, 1
        mov edi, ps
        mov esi, v
        movd mm5, edi
        movd mm7, eax
        add edi, 16

main_loop:

        movd ecx, mm5
        cmp edi, ecx
        je done

        sub edi, 16

        mov edx, [edi + 4]  // memory read
        movq mm6, [edi + 8]  // memory read
        mov eax, [edi]      // memory read

tail:
        cmp edx, 1
        jl main_loop
        jne if_r_eq_n

        movq mm2, mm7
        lea ecx, [eax * 8 + esi]
top:
        por mm6, mm2
        movntq [esi], mm6  // memory write - write-through
        add esi, 8
        pxor mm6, mm2
        cmp esi, ecx
        psllq mm2, 1
        jnz top

        jmp main_loop
if_r_eq_n:

        cmp edx, eax
        jne else_case

        movq mm1, mm7
        movd mm0, eax
        psllq mm1, mm0
        psubq mm1, mm7
        por mm1, mm6
        movntq [esi], mm1 // memory write - write-through
        add esi, 8

        jmp main_loop

else_case:

        dec eax
        mov [edi], eax  // memory write
        movq mm1, mm7
        mov [edi + 4], edx  // memory write
        movd mm0, eax
        dec edx
        movq [edi + 8], mm6  // memory write
        psllq mm1, mm0
        add edi, 16
        por mm6, mm1

        jmp tail
done:
        emms
    }
}
```

This version was still not competitive with Graham Hazel's killer entry. One seemingly obvious optimization is to bundle up 128-bit double-quadwords (SSE data types) and write them when possible with movntdq. The housekeeping this requires made it slower. It's also necessary to maintain 128-bit alignment when using movntdq and that's a pain.

An Old Technique

Loop unrolling as an optimization technique seems to be falling out of favor these days with good branch prediction. For us, it's a keeper. Using preprocessor macros to keep it slightly less ugly, the following code unrolls the inner loop 16 times. Why 16? Trial and error – 16 performed better than 4, 8, 12, 20, 24 and 32. The final MMX version clocks in at around ~1.25ms (7.2-7.5 cycles per element). We'll see how it stacks up against Graham's head to head. It seems to execute slightly faster than Jim's SSE memset. While minor speed improvements may be possible (on the order of tenths of milliseconds), I'm convinced that there's no way to get significantly faster performance.

I could be wrong.

```
////////////////////////////////
// Final mmx version..

#define _ELEM_COMMON           \
} __asm { por mm6, mm2 }       \
  __asm { movntq [esi], mm6 } \
  __asm { add esi, 8 }         \
  __asm { pxor mm6, mm2 }      \
  __asm { cmp esi, ecx }       \
  __asm { psllq mm2, 1 }
#define _ELEM      _ELEM_COMMON __asm { je end_top
#define _ELEM_LAST _ELEM_COMMON __asm { jne top

void Choose64_4_Senzee_mmx_fast(unsigned __int64 *v)
{
    __declspec(align(16))
    struct { int nr, _; unsigned __int64 h; } s[4] = { { 0x4004, 0, 0 } };
    void *ps = &s[0];
    __asm
    {
        mov eax, 1        // registers
        movd mm7, eax     // ---------
        mov edi, ps       // eax = q.n/q.r    ebx = <don't use>
        mov esi, v        // ecx = scratch    edx = pstack base
        mov edx, edi      // edi = pstack     esi = v
        add edi, 16       // mm7 = 1          mm6 = q.h
        xor ecx, ecx

main_loop:

        cmp edi, edx
        je done

        sub edi, 16

        mov eax, [edi]
        movq mm6, [edi + 8]

tail:
        cmp al, 1
        jl main_loop
        jne if_r_eq_n

        movq mm2, mm7
        mov cl, ah
        lea ecx, [ecx * 8 + esi]

top:    _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM
        _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM _ELEM_LAST // unrolled loop

end_top:

        xor ecx, ecx
        jmp main_loop

if_r_eq_n:

        cmp al, ah
        jne else_case

        movq mm1, mm7
        mov cl, ah
        movd mm0, ecx
        psllq mm1, mm0
        psubq mm1, mm7
        por mm1, mm6
        movntq [esi], mm1  // write-through
        add esi, 8

        jmp main_loop
else_case:

        dec ah
        mov [edi], eax
        movq mm1, mm7
        mov  cl, ah
        movd mm0, ecx
        dec al
        movq [edi + 8], mm6
        psllq mm1, mm0
        add edi, 16
        por mm6, mm1

        jmp tail
done:
        emms
    }
}
#undef _ELEM
#undef _ELEM_LAST
#undef _ELEM_COMMON
```