Writing Performant C++ Code

The road to slow code is paved with wrong assumptions

About Me

Larry Bank

I optimize other people's code for a living

Web: https://www.bitbanksoftware.com
Email: larry@bitbanksoftware.com
Twitter: @fast_code_r_us
Github: bitbank2
Blog: https://bitbanksoftware.blogspot.com

Have empathy for your C++ compiler

This might sound silly on the surface, but it just means to try to see your code from the point of view of the compiler.

- Your code tells a story of how you want to process your data
- Providing more detail can create a richer story
- Understanding how your 'verbs' affect the code might cause you to choose different ones
- Provide too little info and the compiler will fill in the blanks with choices you may not appreciate

Know the cost of your choices

One line of code is not equivalent to another; choose wisely.

`A = B + C;` is very different from `memcpy(pA, pB, iLen);`

The first can usually turn into 1 instruction while the second can take any number of cycles to complete. Modern compilers are smart enough to fix silly mistakes like this:

```
memcpy(dest, src, 4);
```

What does the compiler know?

We have to balance a lot of information to accomplish any task. This collection of info (and assumptions) doesn't always translate well into code. Constants versus variables and volatility are some of the most important info that can get 'lost in translation'. An example:

```
int iLen;
const int LEN=4;

  memcpy(pDst, pSrc, iLen); vs memcpy(pDst, pSrc, LEN);
```

In the first example, if iLen == 4, the code will call memcpy() and spend hundreds of cycles copying 4 bytes while in the second, the compiler can probably turn it into a single instruction.

The slowness of OOP

- The compiler treats writes to member variables as if they're volatile
- Methods that modify member variables repeatedly will waste many cycles
- Worst case scenarios (read/modify/write) will occur frequently
- No middle ground between const and normal variables
- Maybe the next version of C++ should include non_volatile as a modifier

Accessing member variables through an object pointer will generate slow code because the compiler can't know how volatile the variables actually are. It will write new values to memory as soon as they change.

```
typedef struct mystuff
{
    int iSum;
} MYSTUFF;

void slow_way(MYSTUFF *pMS, uint32_t *pData, int iLen)
{
    pMS->iSum = 0;
    for (int i=0; i<iLen; i++)
    {
        pMS->iSum += pData[i];
    }
} /* slow_way() */
```

The auto-vectorizer won't even touch it. The best the compiler can do is unroll the loop because it's compelled to write the new value to the member variable as it changes.

```
.LBB0_8:                                # =>This Inner Loop He
        add     ecx, dword ptr [rsi + 4*rax]
        mov     dword ptr [rdi], ecx
        add     ecx, dword ptr [rsi + 4*rax + 4]
        mov     dword ptr [rdi], ecx
        add     ecx, dword ptr [rsi + 4*rax + 8]
        mov     dword ptr [rdi], ecx
        add     ecx, dword ptr [rsi + 4*rax + 12]
        mov     dword ptr [rdi], ecx
        add     rax, 4
        cmp     rdx, rax
        jne     .LBB0_8
        test    r8, r8
        je      .LBB0_6
```

If you're going to make multiple changes to a member variable, it's much quicker to reserve a local variable to do the work and write it into the structure once you've finished.

```
void fast_way (MYSTUFF *pMS, uint32_t *pData, int iLen)
{
    int32_t iLocalSum = 0;
    for (int i=0; i<iLen; i++)
    {
        iLocalSum += pData[i];
    }
    pMS->iSum = iLocalSum;
} /* fast_way() */
```

For this case, the auto-vectorizer can do a decent job and is free to wait until the end of the loop before it has to write the updated value into the structure.

```
        movdqu  xmm2, xmmword ptr [rsi + 4*rax]
        paddd   xmm2, xmm0
        movdqu  xmm0, xmmword ptr [rsi + 4*rax + 16]
        paddd   xmm0, xmm1
        movdqu  xmm1, xmmword ptr [rsi + 4*rax + 32]
        movdqu  xmm3, xmmword ptr [rsi + 4*rax + 48]
        movdqu  xmm4, xmmword ptr [rsi + 4*rax + 64]
        paddd   xmm4, xmm1
        paddd   xmm4, xmm2
        movdqu  xmm2, xmmword ptr [rsi + 4*rax + 80]
        paddd   xmm2, xmm3
        paddd   xmm2, xmm0
        movdqu  xmm0, xmmword ptr [rsi + 4*rax + 96]
        paddd   xmm0, xmm4
        movdqu  xmm1, xmmword ptr [rsi + 4*rax + 112]
        paddd   xmm1, xmm2
        add     rax, 32
        add     rdx, 4
```

Know your target machine

Code which is aware of your target machine's capabilities can make a huge difference in performance. Some of the features to think about:

- Speed disparity between memory and instruction execution
- Memory cache / special regions
- ISA: Floating point / SIMD / hardware divide / special instructions
- Protected mode OS / bare metal

Not all memory is created equal

Is your code running on a PC or embedded microcontroller? This can make a huge difference in how you design your project.

- Embedded CPUs can usually access on-chip SRAM and FLASH in 1 clock cycle
- PC CPUs can have > 200x disparity between SDRAM speed and CPU cycles
- malloc/free evicts data from the cache too - very expensive on a PC, no big deal on a MCU
- Place your working data set in the fastest memory/cache you can

Portable and Target aware?

- Code portability and utilizing unique features of your target machine don't have to be mutually exclusive.
- Only a small portion of your code contains time-critical functions
- Keep a generic C++ reference version of all functions and #ifdef or use templates to coexist with optimized ones for your target platform(s)
- Use CPU intrinsics + SIMD if you really need the speed

Let's talk math...

- Division/Modulus (int and float) is expensive - **always**
  For a circular buffers, avoid modulus with compare+subtract
  For hash tables, make the size a power of 2 and use AND

- Carelessly converting between float and int is expensive

- Math functions are even more expensive if you use the default (double-precision) versions instead of the 'f' 32-bit float (e.g. sqrt vs sqrtf)

- Yes, lookup tables are still a useful tool to avoid repeated calculations

System calls

- Calls into a protected operating system (e.g. fwrite) have a high overhead
- Do as much work per call as possible (e.g. work with files in large chunks)

Slow

```
int my_array[MY_SIZE];
for(i=0; i<MY_SIZE; i++) {
  fread(&my_array[i], sizeof(int),
  1, h);
}
```

Fast

```
int my_array[MY_SIZE];
fread(my_array, sizeof(int),
  MY_SIZE, h);
```

- Manage your own buffers and threads (aka memory / thread pools)

A Brief word on SIMD / Vector code

- SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata
- Special CPU instructions that work with wide registers (e.g. 128-512 bits)

Why should I care?

- SIMD can accomplish much more work per clock than scalar instructions (e.g. Intel AVX can do 8x as many floating point operations per instruction)
- If you don't use them, you're wasting your computer's time and energy
- You normally have to opt-in (compiler flags) to enable them

Auto-Vectorization? Addirittura?

It tries to turn your C++ code into SIMD instructions

This is one of those topics that really irks me because people attribute magical powers to auto-vectorizers, but they're usually very limited in what they can accomplish without understanding their limitations. Here are 2 examples:

- It can work with 'plain' C++ code in a limited way, but works considerably better when you give it some hints

- It usually gives up when there is anything interfering with its starting point of an ideal data loop

Given a simple loop with no info about the value for iLen, Clang generates scalar code and tests for doing the work with SIMD on groups of 8 values at a time. This example was created with Compiler Explorer. You can try it with the following URL:

https://godbolt.org/z/YFq6Mm

```
uint32_t vector_sum(uint8_t *pSrc, int iLen)
{
uint32_t  sum = 0;
    for (int i=0; i<iLen; i++) {
        sum += *pSrc++;
    }
    return sum;
}
```

This conservative SIMD code improves the performance a bit over scalar code. It works with 2 x 32-bit words at a time and interleaves enough for dual-issue execution.

```
.LBB0_7:                                # =>This Inner Loop Hea
        movd    xmm3, dword ptr [rdi + rax] # xmm3 = mem[0],zer
        movd    xmm4, dword ptr [rdi + rax + 4] # xmm4 = mem[0]
        punpcklbw       xmm3, xmm1      # xmm3 = xmm3[0],xmm1[0
        punpcklwd       xmm3, xmm1      # xmm3 = xmm3[0],xmm1[0
        paddd   xmm2, xmm3
        punpcklbw       xmm4, xmm1      # xmm4 = xmm4[0],xmm1[0
        punpcklwd       xmm4, xmm1      # xmm4 = xmm4[0],xmm1[0
        paddd   xmm0, xmm4
        add     rax, 8
        cmp     rdx, rax
        jne     .LBB0_7
        paddd   xmm0, xmm2
        pshufd  xmm1, xmm0, 78          # xmm1 = xmm0[2,3,0,1]
        paddd   xmm1, xmm0
        pshufd  xmm0, xmm1, 229         # xmm0 = xmm1[1,1,2,3]
        paddd   xmm0, xmm1
        movd    eax, xmm0
        test    r8d, r8d
        je      .LBB0_9
.LBB0_4:
```

In this version, we give Clang enough info to know that we're working with a large data set and it can unleash a wider vector loop on the data. Various values of VECTOR_GROUPING work; I used 32 for this example. It now uses the 128-bit registers efficiently and reduces stalls by enabling quad-issue operations.

```
uint32_t vector_sum2(uint8_t *pSrc, int iLen)
{
uint32_t sum = 0;
int i = 0;

    for (; i<iLen-VECTOR_GROUPING-1; i+=VECTOR_GROUPING) {
        for (int j=0; j<VECTOR_GROUPING; j++) {
            sum += *pSrc++;
        }
    }
    // pick up any stragglers
    for (; i<iLen; i++) {
        sum += *pSrc++;
    }
    return sum;
}
```

```
.LBB1_3:                                # =>This Inner Loop Hea
        movdqu  xmm9, xmmword ptr [rdi + rcx]
        movdqu  xmm8, xmmword ptr [rdi + rcx + 16]
        movdqu  xmm0, xmmword ptr [rdi + rcx + 32]
        movdqu  xmm1, xmmword ptr [rdi + rcx + 48]
        movdqa  xmm11, xmm9
        punpckhbw       xmm11, xmm10    # xmm11 = xmm11[8],xmm1
        movdqa  xmm3, xmm11
        punpcklwd       xmm3, xmm10     # xmm3 = xmm3[0],xmm10[
        movdqa  xmm6, xmm0
        punpckhbw       xmm6, xmm10     # xmm6 = xmm6[8],xmm10[
        movdqa  xmm4, xmm6
        punpcklwd       xmm4, xmm10     # xmm4 = xmm4[0],xmm10[
        paddd   xmm4, xmm3
        movdqa  xmm12, xmm8
        punpckhbw       xmm12, xmm10    # xmm12 = xmm12[8],xmm1
        movdqa  xmm3, xmm12
        punpcklwd       xmm3, xmm10     # xmm3 = xmm3[0],xmm10[
        movdqa  xmm7, xmm1
        punpckhbw       xmm7, xmm10     # xmm7 = xmm7[8],xmm10[
        movdqa  xmm5, xmm7
        punpcklwd       xmm5, xmm10     # xmm5 = xmm5[0],xmm10[
        paddd   xmm5, xmm3
        paddd   xmm5, xmm4
        punpcklbw       xmm9, xmm10     # xmm9 = xmm9[0],xmm10[
        movdqa  xmm3, xmm9
        punpcklwd       xmm3, xmm10     # xmm3 = xmm3[0],xmm10[
        punpcklbw       xmm0, xmm10     # xmm0 = xmm0[0],xmm10[
        movdqa  xmm2, xmm0
        punpcklwd       xmm2, xmm10     # xmm2 = xmm2[0],xmm10[
        paddd   xmm2, xmm3
        punpcklbw       xmm8, xmm10     # xmm8 = xmm8[0],xmm10[
```

Auto-Vectorizer Limitations

```
#define VEC_SIZE 16

uint32_t vector_sum(int32_t *pA, int32_t *pB, int iLen)
{
uint32_t  sum = 0;

    for (int i=0; i<iLen-VEC_SIZE-1; i+=VEC_SIZE) {
        for (int j=0; j<VEC_SIZE; j++) {
        if (pA[i+j] >= 0)
            sum += pA[i+j] * pB[i+j];
        }
    }
    return sum;
}
```

The compiler can deal with certain conditional statements within the main loop, but usually gives up when individual elements of a SIMD register are treated differently. Hand-written SIMD code can do an efficient job of this by using compare+mask operations, but Clang is only able to optimize it by unrolling a scalar loop.

SIMD / Vector Challenges

- Too much software never makes use of these powerful instructions because it takes extra effort to enable or add them to your code
- Your algorithm needs to allow for operations to occur in parallel. Imaging/pixels are usually good candidates
- Compilers are not very good at creating SIMD automatically. This means you may need to add intrinsics to your code to explicitly tell the compiler your intent.
- Each platform has its own unique SIMD instructions. The implementations are usually similar, but custom code must be written to take advantage of special features in each platform.
- Your data doesn't always fit neatly into the SIMD register size, but with some creative thinking you can still take advantage of SIMD instructions

Rules of the Road

Programmers who write efficient code...

- Do their work in local/register variables
- Avoid dynamic structures in inner loops
- Avoid using too many nesting levels
- Choose variable types and mathematical precision carefully
- Minimize the number of integer and float conversions
- Minimize the number of OS calls
- Minimize the use of divides and trigonometric functions
- Make use of SIMD and target-specific features when possible
- Know the nature and statistics of their data
- Test boundary conditions only on the boundaries
- ***** Test every assumption *****