As an introduction to low level optimization, we try to measure the cost of a single line of code. Running on a modern processor, and compiled by a modern compiler, a single line cannot be evaluated without context. We explore CPU architectures and inspect the output of the compiler. We conclude our foray with a discussion of data types and an overview of common techniques that improve application performance, the “Rules of Engagement”.

Cost of a Multiply

Let’s start with a simple experiment in which we try to measure the cost of a single multiplication in C:

```c
0  starttimer();
1  float x = 0;
2  for( int i = 0; i < 50000; i++ ) x *= y;
3  stoptimer();
```

Although this looks innocent enough, it does not yield the number that we are looking for. First of all, we measure far more than just the multiplication: there’s the loop (increment, comparison, conditional jump at the end), but also the overhead of the timer and the initialization of variables `x` and `i`. What’s worse though is that the measurement yields ‘0’… The compiler will eliminate the code, because the result of the calculation is not used.

A better way to perform our measurement is to create a context in which we can include or exclude the operation we wish to time. The difference between the executions is the ‘cost’ of the operation. For example:

```c
0  float x = 0, y = 0.1f;
1  unsigned int i = 0, j = 0x28929227;
2  for( int k = 0; k < 50000; k++ )
3      { 
4        x += y, y *= 1.01f;
5        i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
6        if (do_operation) x *= y;
7      }
8  printf( x + (float)i );
```

Line 9 ensures that the code is not eliminated. Line 4 ensures that the operation gets different inputs in each loop iteration. Lines 5 and 7 inject some integer operations, so the loop does not purely consist of floating point work. Line 6 uses a templated bool to toggle our operation.
**x86 in Five Minutes**

To see if this is indeed better code for our purposes, we deep-dive into the code the compiler generates for our snippet.

| PRACTICAL NOTES – In Visual Studio, we can view the assembly code by placing a breakpoint; once we hit the breakpoint, we can simply ‘Go To Disassembly’ using the right-click menu for a source line. |

The generated code is shown on the right.

Chances are that you are not fluent in x86 assembler. Luckily, just reading it isn’t too hard once we know a few basic things about x86-compatible CPUs.

Modern AMD and Intel CPUs are still compatible with Intel’s original 16-bit 8086 processor (1978). These processors used eight registers:

- AX (‘accumulator’), consisting of bytes AH and AL;
- BX (‘base register’), consisting of bytes BH and BL;
- CX (‘counter register’), consisting of bytes CH and CL;
- DX (‘data register’), consisting of bytes DH and DL;
- BP (‘base pointer’), SI and DI (‘source’ and ‘destination index’) and SP (‘stack pointer’).

For the 32-bit 80286 processor, these registers became 32-bit: EAX, EBX and so on, although AX and even AH and AL can still be used to operate on parts of the 32-bit registers. Likewise, 64-bit processors use 64-bit registers: RAX, RBX, ... . Additionally, these processors can use 64-bit registers R8..R15.

Starting with the 80387, floating point numbers are stored in a small stack of 8 registers, named st0..st7.

There is more (SSE / AVX registers etc.), but for now we have enough to interpret the assembler code generated for the loop.

### Assembler Loop

The assembler code starts with a `fldz` command (line 1). This loads value 0.0 to the floating point stack, and corresponds to the `float x=0` command in the C code. The 0.1 value in the code is handled differently: it is loaded, on line 3, from memory address 405290h (hexadecimal). But before that, on line 2, a mysterious operation appears: `xor ecx, ecx`.

Xor’ing a value with itself yields 0: `xor ecx, ecx` is one way to set an integer value to 0. The ecx register thus contains the loop counter 1!
On line 7, register esi is initialized with 0C350h, which is 50000 in decimal.

So far the link between the C code and the assembly is easy to see. Things get interesting on line 8 though. What is value 91D2A969h? The decimal version doesn’t ring a bell... That is, until we see the mul on line 12. The C code performs a division by 28763, and it turns out that 91D2A969h is in fact 246/28763. The compiler replaced a division by a multiplication using the reciprocal of the constant!

**Digest**

Completing the analysis of the assembler code is left as an exercise for the reader. We can already make several observations however:

- The compiler is reordering code to interleave floating point and integer instructions;
- The compiler is optimizing the code.

Question is of course: why does the floating point code get interleaved with integer code? And why is a division replaced by a division? That second question touches on a common guideline in low level optimization: *some instructions take more time than others*. As a rule of thumb:

- Multiplication takes longer than addition and subtraction.
- A division however is much more expensive than a multiplication.
- The real cost however is in square roots, powers and transcendental functions.

This ranking is generally true on any processor, including mobile processors and even GPUs.

**CPU Primer**

Let’s return to the interleaving of integer and floating point code. To execute assembler code, a CPU needs to execute several subtasks: *fetch, decode, execute* and *write-back*. Fetch will read an instruction from memory. The fetched instruction is then decoded and executed. The result of the operation is finally written back, to memory or to a register.

Early processors would execute these steps sequentially, with one step per clock cycle. A 4.77Mhz CPU such as Zilog’s Z80 will thus execute far less than 4.77 million instructions per second. But also: the transistors dedicated to each step are idle for 3 out of 4 cycles.

To improve the efficiency of the ‘fetch-decode-execute’ cycle, later processors operated on instructions in a *pipeline*. At any point in time, multiple instructions are in flight: one is getting fetched, one decoded, and so on. This approach comes at a price: what if one instruction depends on the result on the previous? In such a case, the instruction must be delayed until the previous instruction has completed; we get a *bubble* in the pipeline. But what if the instruction is a *jump*? No problem, we will fill the pipeline with instructions at the address the jump points to. But, what if the jump is *conditional*? Now we have a problem. A modern processor will try to predict the condition, but if it mispredicts, the pipeline must be restarted.
We now have the following pipeline:

Remember that each of the steps uses different silicon. The above model appears to make optimal use of that silicon, but there is a problem: the execute stage. A floating point instruction requires very different execution logic than an integer instruction. To use all parts of the CPU all the time, it makes sense to duplicate the fetch, decode and write-back stages, so that the pipeline can work on an integer and a floating point instruction (and possibly more) in parallel. This is the superscalar pipeline, which is used in most modern processors.

To feed a superscalar pipeline, we need a blend of different instructions. And this is where the compiler comes in: it will try to emit code that runs optimal on the target CPU. This does require of course that our code allows this: if there is no floating point code, that silicon will be idling, no matter how advanced the compiler is.

All Together Now

With the above in mind, we can adjust our insights:

- The goal of low-level optimization is optimal assembler for the CPU.
- Optimal assembler minimizes dependencies between instructions, has predictable conditional jumps and has a blend of different instructions.
- The compiler will try to produce this optimal assembler.
- It does need our cooperation.

It may seem tempting to write optimal assembler manually for critical code sections. In practice this is rarely worth the effort: it requires in-depth knowledge of assembly language and the characteristics of each supported CPU, and results in virtually unmaintainable and unportable code. We can however tweak our C code, inspect the generated assembly, and tweak the source code when the compiler fails to produce efficient assembly.
Conversions

One common source of inefficiency is implicit data type conversion. Here is an example:

```c
struct Color { unsigned char a, r, g, b; 
};
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].r *= 0.5f;
    bitmap[i].g *= 0.5f;
    bitmap[i].b *= 0.5f;
}
```

This code takes a bitmap of 640 by 480 pixels, and halves the intensity of each pixel.

There is a problem with this code: three byte values are promoted to floats, then scaled by 0.5, and finally demoted to unsigned chars to be stored in the bitmap. The resulting assembler is horrible (and slow!):

```assembly
movzx eax, byte ptr [ecx-1]
mov dword ptr [ebp-4], eax
fild dword ptr [ebp-4]
fnstcw word ptr [ebp-2]
movzx eax, word ptr [ebp-2]
or eax, ochOOh
mov dword ptr [ebp-8], eax
fmul st, st(1)
fldcw word ptr [ebp-8]
fistp dword ptr [ebp-8]
movzx eax, byte ptr [ebp-8]
mov byte ptr [ecx-1], al
```

Perhaps that doesn’t seem so bad, but wait: this is just for the red component. The full code for the loop repeats this three times! Can we do better? Certainly:

```c
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].r >>= 1;
    bitmap[i].g >>= 1;
    bitmap[i].b >>= 1;
}
```

This time, the multiplication by 0.5 has been replaced by a bitshift. Shifting an integer one bit to the right halves it (rounding down). This is a pure integer operation, and that shows in the resulting assembler code:

```assembly
shr byte ptr [eax-1], 1
shr byte ptr [eax], 1
shr byte ptr [eax+1], 1
```

This time, the code handles red, green and blue. Needless to say: it’s faster. And we are not even done yet.
Consider the following version of the loop:

```c
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].argb = (bitmap[i].argb >> 1) & 0x7f7f7f;
}
```

Here, we exploit the fact that one pixel consists of four bytes: red, green, blue and alpha. Instead of bitshifting them one by one, we can also bitshift the full 32-bit value. The consequence is that the lowest bit of red will be shifted into green. However, in the final result, this bit must be 0, so we use a bitwise and to remove the top bits of red, green and blue: 7fh is 01111111b (binary). The loop now reduces to two instructions.

**Rules of Engagement**

It can be helpful to use the following *rules of engagement* to get started on low-level optimization. The list is not complete, not always applicable and obviously it should not be used without profiling.

#1. Avoid Costly Operations.

As discussed earlier. Sine, cosine and square roots are red flags. Make sure your profiler confirms this. Optimize by precalculation / look-up tables. Optimize divisions by using the reciprocal when possible.

#2. Precalculate.

“Every optimization is an exercise in caching” a wise man said. Precalculation means: re-using (partial) results, incremental calculations (interpolation, reprojection), but also loop hoisting (pulling constant expressions out of loops) and look-up tables.

#3. Pick the Right Data Type.

A modern processor needs to promote bytes to integers before working with them. Heed implicit conversions. Blend integer and float code to better utilize the pipeline.

#4. Avoid Conditional Branches.

Conditional code is not just if-statements: min(a,b) may also be conditional. Sometimes a loop with a condition can be split in two loops without conditions. Sometimes a look-up table can prevent a condition. Unrolling a for-loop reduces the number of conditional statements. And if all else fails: at least make the conditions predictable.

#5. Early Out.

A loop can often be terminated early. Same for a function.

#6. Use the Power of Two.

Bit-operations are very efficient. A division by a power of 2 is a bitshift, and so is a multiplication by a power of 2. This is particularly important for 2D arrays: accessing element
(x,y) involves a multiplication by the width of the table. If this width is a power of 2, every array access is faster. Make sure you know the powers of 2 (1..16) by heart.

#7. Do Things Simultaneously.

Use the cores. An integer holds four bytes; use them simultaneously.

Once more I emphasize that these rules should be applied after studying the output of a profiler. Don’t go blindly after a square root in an initialization function.

The End

You can find an example of low-level optimization applied to a practical application here.

In subsequent lectures we will look at the hardware in more detail. A lot of attention will go to the vector hardware in the CPU and the memory system. Large gains can be achieved by adapting code to these components.

This material is part of the Optimization & Vectorization course of the MGT master program at the Utrecht University in the Netherlands. More information at:

http://www.cs.uu.nl/docs/vakken/mov.

Utrecht University