Sweaters For Me...

Look-Up Tables (LUT) Operations in C++

Carlos Moreno

Introduction

Look-Up Tables are a technique commonly used to accelerate numeric processing in applications with demanding timing requirements. They are often used in Image Processing and DSP (Digital Signal Processing) applications.

The basic idea is to pre-compute the result of complex operations that are or can be expressed as a function of an integer value. The pre-computed results are typically stored in an array, which is used at run-time instead of performing the whole, time-consuming operation.

In this article, I present an implementation in C++ of a wrapper class to provide LUT operations in a way that is completely transparent to the user of that class. I first introduce the concept and discuss the benefits and implications of their use, and then describe the implementation.

The article requires previous knowledge of the basic tools of the C++ language, in particular, class templates and operator overloading.

What is a Look-Up Table?

Simply put, a Look-Up-Table (LUT) is an array that holds a set of pre-computed results for a given operation. This array provides acces to the results in a way that is faster than computing each time the result of the given operation.

LUT's are typically used in real-time data acquisition and processing systems (often embedded systems), since these types of systems impose demanding and strict timing restriction. An important detail to consider is the fact that LUT's require a considerable amount of execution time to initialize the array (to pre-compute the results). In real-time systems, it is in general acceptable to have a delay during the initialization of the application (after all, the application will be presumably run right after boot, which takes a few seconds anyway).

Let's take a look at a simple example to illustrate the idea of LUT's. Consider a real-time data acquisition and processing system in which data are sampled as 8-bit numbers, representing positive values from 0 to 255. Suppose that the required processing involves computing the square root of every sample. In C, the use of a LUT to compute the square root would look as follows:

```
double LUT_sqrt [256];     /* presumably declared globally */

 /* Somewhere in the Initialization code */

for (i = 0; i < 256; i++)
{
               LUT_sqrt[i] = sqrt(i);/* provided that <math.h> was #included */
      }
```

```
/* During the normal execution of the application */

result = LUT_sqrt[sample]; /* instead of result = sqrt(sample); */
```

During the initialization phase, the application sacrifices a certain amount of time to compute all the 256 possible results. But the key point is that after that, when the system starts to read data in real time, the system will be able to complete the processing required in the time available. Remember that data must be read at fixed intervals of time, and the processing of one datum must be completed before the interval of time until the next datum expires (otherwise, the system would collapse).

Can LUTs be Used for Operations on Non-integer Arguments?

Despite the obvious, the short answer is yes.

Given that the results are stored in an array, and the array subscript must be an integer number, it would look like LUT's can only replace operations that are function of an integer argument. Being a little more specific, we should say that they can only replace operations that *can be expressed as a function of an integer argument*.

A typical example of this is found in some common Digital Signal Processing techniques, in which we often need to evaluate the following functions:

sin (2πk/N) and cos (2πk/N)

where N is a constant, and k = 0,1,2, ... N-1.

The argument of the functions sin and cos are obviously not integers. But the entire function can be described as functions of the integer argument k.

Thus, we could use a LUT to evaluate those functions as shown in the example below:

```
#define pi 3.1415926536 /* or const double pi = 3.14... , if C++ */

double LUT_sin_2pi_N [N], LUT_cos_2pi_N [N];

      /* Somewhere in the Initialization code */

for (k = 0; k < N; k++)
{
  LUT_sin_2pi_N[k] = sin (2*pi*k/N);  /* provided that <math.h> was #included */
  LUT_cos_2pi_N[k] = cos (2*pi*k/N);
}

/* During the normal execution of the application, if we need to compute
x[k] * sin(2πk/N), we use the following: */

result = x[k] * LUT_sin_2pi_N [k];
```

In this case, we are using only one access to a memory element, instead of two multiplications, one division and one evaluation of a math function.

This example is even nicer if we use C++'s standard library class complex to write code that better follows the standard math notation for these DSP techniques (using complex number exponentials, instead of trigonometric functions). We are in general interested in evaluating functions similar to:

$e^{j2\pi k/N}$

where *j* is the imaginary unit.

The example in C++ to implement this could be as follows:

```
typedef complex<double> Complex;

const double pi = 3.1415926536;
const Complex j(0,1);  // Imaginary unit

Complex LUT_e_j2pi_N [N];

// Somewhere in the Initialization code
for (k = 0; k < N; k++)
{
   LUT_e_j2pi_N[k] = exp (j*2*pi*k/N);
}

// During the normal execution of the application, if we need to compute
// x[k] * exp(j*2*pi*k/N), provided that 0 <= k < N, we use the following:
Complex result = x[k] * LUT_e_j2pi_N [k];
```

In the examples presented so far, I have used LUT's only for operations expressed as a function of positive integer arguments. That is not a restriction, as the following example shows:

```
    typedef complex<double> Complex;

    const double pi = 3.1415926536;
    const Complex j(0,1); // Imaginary unit

    Complex LUT_space[2*N - 1];
    Complex * const LUT_e_j2pi_N = LUT_space + N - 1;

    // Somewhere in the Initialization code
    for (k = -(N-1); k < N; k++)
    {
      LUT_e_j2pi_N[k] = exp (j*2*pi*k/N);
    }

    // During the normal execution of the application, if we need to compute
    // x[k] * exp(j*2*pi*k/N), for any k such that -N < k < N, we use the following:

    Complex result = x[k] * LUT_e_j2pi_N [k];
```

Encapsulating These Ideas

It would be convenient to encapsulate the LUT related data and functionality. There are several reasons for this:

1. There are several pieces of information (several data items) that conceptually represent one LUT.
2. There is an initialization process that must be always done.
3. The syntax to obtain the result of an operation is not transparent -- it requires an access to an array, which shows the trick that is being used, instead of what the program is actually doing.

Item 1 suggests that encapsulation is appropriate to provide a single-object representation of the LUT. Item 2 suggests that we may take advantage of the class constructor to automatically initialize the object without us having to always keep in mind that we need to rewrite that section of code. And finally, item 3 suggests that we may take advantage of the operator overloading feature to provide a syntax that is transparent to the clients of this class.

I am now ready to present the class definitions for the two types of LUT's: LUT's to perform multiplication of an integer number times a floating-point number, and LUT's to evaluate functions.

A Class to Represent Floating-Point Multiplication LUT's

This may sound a little strange. Why would we need a LUT to pre-compute the results of an operation so simple as multiplication of two numbers? Floating-point operations, in particular multiplications, have a high cost associated in terms of processor time. Recent processors that feature an embedded floating-point unit (e.g., Intel x486 and above) perform these operations very efficiently -- maybe as fast as a single access to a variable in memory! In these cases, LUTs would be completely useless: they will not accelerate multiplications by floating-point numbers. But in older processors, like Intel x386, x286, and x86/x88, floating-point operations are much slower than a memory access. It is not uncommon to find single-board PC's for embedded applications that are based on these processors. Embedded applications is precisely the type of application that most commonly requires real-time processing with demanding timing restrictions, which makes it reasonable to think about using LUT's to accelerate multiplications by floating-point numbers.

The class is defined as a template with a type argument to specify the result of the multiplication, two non-type integer arguments to indicate the range of the integer operand, and a non-type argument to specify the scaling factor in the multiplication. The scaling factor is useful when we want to have the result as an integer type with increased resolution. This allows us to perform all the arithmetic operations in integer format with a considerably reduced rounding-error problem (if we use a scaling factor large enough).

Listing 1 shows the class definition and implementation for class LUT_number (member functions are inlined, given that the main goal of LUT's is run-time efficiency).

The examples below illustrate the use of this class to smooth a sequence of values by applying a moving weighted average filtering (i.e., for each value in the sequence, the output of the operation is the weighted average between that value and the four values surrounding it).

```
#include "lut.h"
// ... other definitions
signed char x [BUFFER_SIZE];

typedef LUT_number<-128, 127, long int, 256> coeff;

const coeff Ho = 0.6;
const coeff H1 = 0.3;
const coeff H2 = 0.1;

signed char moving_avg_filter (const signed char * x)
{
   return (Ho*x[0] + H1*x[1] + H1*x[-1] + H2*x[2] + H2*x[-2]) / 256;
}
```

Notice that the operations are evaluated that way to avoid overflow problems (e.g., `H1*x[1] + H1*x[-1]` instead of `H1 * (x[1]+x[-1])`).

The whole operation is done without performing a single floating-point operation (neither multiplications nor additions). Furthermore, the optimizer could decide to do the division by 256 by shifting eight bits, which is much more efficient.

A Class to Represent Function Evaluation LUT's

In this case, the class is defined as a template class taking a type parameter specifying the result type of the operation, a type parameter specifying the data type of the argument of the function that is being "emulated" by the LUT, and two non-type parameters specifying the range of the integer argument. The class provides an overloaded `()` operator taking one integer as parameter to allow clients of the class to use a syntax equivalent to that of evaluating a function. Furthermore, the constructor receives, as argument, a pointer to the function and an optional parameter indicating the factor that multiplies the argument of the function.

Listing 2 shows the definition and implementation of class LUT_function. The header file lut.h shows both template class definitions.

The example below shows how to use this class to evaluate the functions f(k) = sin(2pik/N) and g(k) = cos(2pik/N).

```
#include <cmath>
const double pi = 3.1415926536;

#include "lut.h"

LUT_function<0, N> Sin (sin, 2*pi/N),
Cos (cos, 2*pi/N);

// ...
// Somewhere in the code:

for (int k = 0; k <= N; k++)
{
  total += x[k] * Sin (k) + x[-k] * Cos (-k);
}
```

Use LUTs with Care

You have to be very careful before deciding to use LUT's to solve a particular problem. We are often adicted to tools or tricks that seem to provide faster execution. An example of this is the use of inline functions in C++, or the use of macros in C.

Before using LUT's, we should ask ourselves (and try to find an accurate answer!) the following two questions:

1) Is there a high cost to pay in exchange for speed? (and if there is, is it worth paying that cost?)

and

2) Is a LUT really going to accelerate the processing?

The first question seems reasonable, and relatively easy to evaluate. For example, if we need to store pre-computed results (of type double) for an operation that takes a 16-bit integer as an argument, we better think twice before using the LUT! We would require a table holding 65536 elements of type double! (8 bytes each).

What is the answer in a situation like that? It depends on many factors. I can not give a general answer or recipe. You have to evaluate and prioritize your resources and requirements given for the specific application.

But the second question seems stranger. LUT's are intended to (and supposed to) accelerate certain time consuming operations. But that may not always be the case. In particular, it is not obvious that reading a number from memory will be faster than performing one floating-point multiplication. For processors featuring a hardware floating-point processing unit, it is very likely that performing the multiplication will be faster. Keep in mind the following: processors feature a clock multiplier for the internal hardware of the chip. This means that if the external circuits (including system memory!) are run at 100 MHz, internal hardware (on-chip circuitry) may be running at 400 or 500 MHz. Accessing system memory is done at a clock speed that is 5 times slower than the clock speed controlling the internal operations (including multiplications). Of course, the multiplication involves more steps than reading from memory, but on the other hand, there is a specialized hardware section inside the processors to efficiently perform floating-point arithmetic operations. The overall result is that floating-point multiplications are in general faster.

That is not the case for older processor architectures. In particular, for Intel x386 and older architectures, it is almost certain that the use of LUT's will tremendously accelerate multiplications. The use of these processors is very common in embedded systems (e.g., computer based control, monitoring, data acquisition and processing, etc.), given that there are many single-board PC computers at prices that make them very attractive to develop devices that require software-based operations.

In short, you should think carefully before using LUT's to solve a particular problem. I would even dare to recommend that, in general, you test and measure the performance of both methods before making a decision.

Avoid Non-Static LUT Objects

Another detail to be careful about is the fact that LUT's require a long initialization process, and take a lot of memory. Thus, you should always try to use them only as global or local static objects. That way, they are initialized at startup, and are always ready to use. Also, we avoid using stack space to store the (possibly huge) array of pre-computed results.

Another detail that may be very obvious, but I will mention it anyway: never instantiate on-the-fly or cast to LUT objects to accelerate an individual operation in the middle of your code! (notice that implicit type conversion is disallowed in the class definition, using the **`explicit`** attribute in the constructor).

Conclusions

Look-Up Tables (LUT's) are a convenient tool to accelerate operations that can be expressed as functions of an integer argument. They use stored pre-computed results that allow the program to immediately obtain a result without performing again the time-consuming operation.

However, we have to be very careful before deciding to use LUT's to solve a particular problem. First of all, we should carefully evaluate the cost associated to their use (in particular, the memory space that will be required to store the pre-computed results). And secondly, we should keep in mind that is not always the case that LUT's accelerate the process. Depending on the processor that we are using, it may not be the case (in particular, for recent processors that perform very efficiently operations on floating-point numbers, it is often inappropriate the use of LUT's).

In embedded systems, where it is common to find older processor architectures or microcontrollers, the use of LUT's is often appropriate.

The use of C++ to implement this tool turns out to be appropriate. Without any sacrifice in run-time efficiency, I presented a highly reusable class definition that provides the required LUT operations in a very transparent way, making it easier to develop data processing algorithms in an efficient way.

Bibliography

The C++ Programming Language, 3rd Edition. Bjarne Stroustrup.
Addison-Wesley, 1997.

Main Page Articles Tutorials