

Getting an algorithm ready for reuse

By Ketul Patul, [EE Times](#)

November 02, 2004 (11:22 AM EST)

URL: <http://www.eetimes.com/article/showArticle.jhtml?articleId=51202101>

Embedded-system designers must reuse not just hardware intellectual property but software as well. Often this is not a simple matter of recompilation. Software must be designed specifically for reuse. Ironically, this is often achieved using more hardware-specific languages and techniques. This article will discuss the optimization techniques we used for making a pattern-matching and image-processing algorithm reusable on a Texas Instruments Inc. DSP.

This C++ algorithm ran on a PC platform, and the ultimate goal was to develop a handheld medical-imaging device based on an embedded platform. The algorithm identifies specific features of the human face and displays the modified face image on the LCD of the handheld medical device. A TMS320C6205 provided enough horsepower to execute the algorithms, but the major challenge was how to deal with the available 64 kbytes of internal data memory.

Everyone assumes that algorithms written in C++ are easily portable across platforms. This is not true in practice. On our first attempt to port the algorithm, we recompiled the code on Texas Instruments' Code Composer Studio. But when we ran the code on the DSP, it took five seconds to execute against the target execution time of 100 milliseconds. This 50x slowdown was attributed to the algorithm's being designed for PC processors.

To make the algorithm reusable, the first step was to completely analyze it and identify the optimization areas. The major areas were C++ to C conversion; efficient use of direct memory access (DMA) control and memory; logic optimizations like optimizing the C code; and floating-point to fixed-point operations.

The conversion was carried out in these steps and at the end of each stage the algorithm was tested for functionality and performance. The optimal execution time was achieved using only C code and did not require any DSP assembly code.

Porting to C

When porting C++-based architecture to C, we evaluated the complexity of the class definitions and their run-time usage, broke the class hierarchy and constructed simplified structures. To achieve real-time performance on a DSP, we avoided hierarchical structure referencing. The methods within the C++ classes were converted to global functions to operate on the C data structures. The original algorithm used vectors throughout. To avoid major changes in the algorithm, the vector functionality in C was developed with the help of hash tables. The C++ templates degraded performance, so they were removed.

The original algorithm needed 48 kbytes of memory for the stack and 139 kbytes for the program. To reduce the stack usage, we reduced parameter passing in functions by using global variables. This reduced the stack size of the algorithm to a mere 3 kbytes, a 94 percent reduction in size. The program memory was reduced to 70 kbytes by removing C++ overheads and eliminating redundant functionalities. Execution and verification of this stage was done on the PC platform for faster execution and debugging. This stage of optimization is totally platform-independent and can be applied to port code on any embedded platform.

In PC-based algorithms, all data referencing and data transfer is done conventionally, using "malloc" and "memcpy" functions. In new-generation DSPs, DMA is available and it is an overkill to use the CPU for data transfer. We removed run-time memory allocation, using static allocation instead, and also developed wrapper functions for "memcpy" and "memset" to use DMA instead of memory library functions.

The algorithm operated repetitively on the image pixel by pixel. The same code would work on the DSP in an optimized way, provided the image data is stored in the internal data memory. Since the DSP has only 64 kbytes of internal data memory available, all the intermediate images were stored in external SDRAM. It takes about five clock cycles to access data from internal memory, and about 16 clock cycles for external memory. As intermediate images are stored in the external memory, pixel-by-pixel processing of the images took too much time. So instead of accessing a single pixel, processing it and copying back to the memory, we got the multiple lines into the internal memory via DMA. Using DMA to copy a chunk of data is much more beneficial than using a CPU. We happened to modify each and every FOR loop of the code to modify it to process line by line. After processing each line, we moved the processed line data back to external memory using DMA.

Still, the DMA transfer was not fully utilized, since intermediate images were not 32-bit aligned. Because intermediate images were extracted from the input image based on the region of interest, the starting pixel of intermediate images was not 32-bit address-aligned every time. So we resized the region of interest to align it along a 32-bit boundary, thus allowing us to use 32-bit DMA transfers. This stage is also platform-independent so long as the target embedded processor supports DMA transfer.

Optimizations in C code

First we identified the time-consuming sections by breaking the whole algorithm into smaller units. The more time-consuming functions were rewritten and optimized. For example, we used register variables instead of global and simplified the FOR loops, allowing the compiler to do more instruction parallelism and pipelining. In some places we merged two FOR loops that were operating on the same input data to reduce the data copy overheads. We removed small function calls and in-lined them, increasing the program size a bit but getting a good performance improvement.

In this stage of optimization, we measured code performance after every change, and undid changes that didn't improve speed. This was an iterative process, but helped in achieving considerable optimization. We did not change the functionality. We just concentrated on the coding methodology and converted the code to optimized C for embedded systems. This stage is almost platform-independent as the code optimization had been done for C code but obviously the compiler-dependent optimizations would need to be reoptimized for other DSPs.

To use the compiler's optimization capability effectively, we removed the floating-point calculations from the FOR/WHILE loops. The resulting code met our objectives. This experience clearly shows that in IP reuse, there may be as much effort in reusing so-called hardware-independent C++ code as in reusing hardware IP blocks.

Ketul Patul (ketul@einfochips.com) is project manager for eInfochips Ltd. (Ahmedabad, India).

```

Original code
void func0()
{
    long k, p, z;
    unsigned char result[100];
    unsigned char i = 0;

    for (j = 100; j-- > 0)
    {
        p = 0.03664;
        z = 0.03662;
        x = 0.5 * (y * (z + 1.0));
        result[i] = (unsigned char) x;
    }
}

Removed floating-point calculations from code
void func0()
{
    long k, p, z;
    unsigned char result[100];
    unsigned char i = 0;

    for (j = 100; j-- > 0)
    {
        p = 3664; // 0.03664 = 3664/100000000
        z = 3662; // 0.03662 = 3662/100000000
        x = 50000 * ((y * (z + 100000)) / 100000000000);
        result[i] = (unsigned char) (x / 100000);
    }
}

Removed division operations from code
void func0()
{
    long k, p, z;
    unsigned char result[100];
    unsigned char i = 0;

    for (j = 100; j-- > 0)
    {
        p = 3664; // 0.03664 = 2*20^7
        z = 3662; // 0.03662 = 2*20^7
        x = 50000 * ((y * (z + 100000)) / 100000000000);
        result[i] = (unsigned char) (x >> 10);
    }
}

```

[See related chart](#)

The original algorithm used a lot of bulky floating-point calculations that, within the CPU cache, prevented the compiler from performing the loop optimization efficiently. Those calculations were removed from the loops, and the resulting code met objectives.



**Wondering what's new
in the memory market?**

[Copyright © 2003 CMP Media, LLC](#) | [Privacy Statement](#)