Utilizing Vector Floating Point of ARM Processors in Compact Framework

by Jaafar Alsalaet

College of Engineering-University of Basrah

1. Introduction

Most of the new ARM processors have Vector Floating Point (VFP) coprocessor which provide hardware implementation of floating point operations. The VFP capability offers the enhanced performance required in some applications such as automotive industry, Digital Signal Processing and filtering, high-speed controllers and many other applications. For example, ARM11 processors have optional VFPv2 as part of ARMv5TE and ARMv6 architecture which can be used as a part of System on Chip (SoC) design. More advanced architecture based on ARMv7 implement VFPv3 as a part of Thumb and ThumbEE instruction set. Two versions of VFPv3 are available; D32 and D16 with 32 and 16 double word registers respectively. Using VFPv3 combined with NEON multimedia processing capability greatly enhance the performance of image processing such as scaling, 2D and 3D encoding/decoding and filtering.

While some of the operating systems, such as Android and some Linux, can utilize the VFP capability of ARM, Windows CE and Compact have restricted access to it especially when developing applications for .NET Compact Framework. This article will shed some light on the utilization of VFP in such environment. Two operating systems will be discussed Windows CE 6 and Windows Embedded Compact 7 with two platforms ARM11 and Cortex-A8.

2. ARM11 with Windows CE 6

A Windows CE 6 platform with S3C6410 processor (ARM11-1176) running at 667MHz is used in this configuration to test the VFP performance. Since Windows CE 6 BSP of the board does not support VFP by default, the ARM VFPv2 subproject is added to the OSDesign project in Windows CE Platform Builder. This subproject generates a replacement FPCRT.dll library which utilizes hardware floating point coprocessor. This is well documented in MSDN and some other sites.

By default, Windows CE 6 platform builder generates the standard Floating Point C Runtime library (FPCRT.dll) which emulates all the floating point operations and transcendental mathematical functions. Applications that wish to use floating point operations and mathematical functions simply call the implemented functions inside FPCRT.dll. Fortunately, the FPCRT library can be replaced by OEM to permit the utilization of VFP.

Another problem arises about the ability of different applications to utilize the updated FPCRT. Generally, there are two types of Windows applications:

1. Managed code which is produced by Just In Time (JIT) complier when compiling .NET compact framework or CLR projects such as projects developed by Visual Basic or C#.

2. Unmanaged code (native code) which is produced at build time by C++ compiler/linker.

The unmanaged code (applications developed by C++) are very efficient in utilizing FPCRT library since both of them are in the same environment (native code). Moreover, the C++ project can be compiled with certain compiler options (will be discussed soon) which forces the compiler to generate VFP instruction to implement floating point operations directly without calling FPCRT functions. On the other hand, managed code suffer from a degraded performance when calling FPCRT functions to implement the required operations. This is due to the fact that managed code is executed in the virtual (non-native) environment and need to perform P/Invoking and marshalling to call native functions of FPCRT.dll or coredll.dll.

Since .NET framework provide great flexibility and capability in software development, most application are developed under its environment. These applications will generally be inefficient in floating point and math calculations. One possible solution to enhance the performance is to mix managed and unmanaged codes in one project. This is possible in full .NET framework by utilizing C++ CLR projects, but it is not available in compact framework. A solution to this problem is to implement the intensive math calculations within the managed code by implementing native DLL functions that are exposed to managed applications by Platform Invocation and data Marshalling. To reduce the latency of P/Invoking and marshalling, it is recommended to issue few calls of bulky calculations over using many calls of tiny calculations. In other words, use unmanaged code for math and managed code for GUI.

To get the best of the unmanaged code DLL, the following hints are found to be useful:

1. Do not use classes, make your project as simple as possible. Classes increase latency during call.

2. It is advisable to define exported functions with extern "C" directive to maintain the same function name without decoration

3. To include VFP instructions in the undamaged code for better performance, use " `/arch:VFPv2 /QRfpe-`" as additional options in compiler command line when compiling C++ DLL project. To do so, you need to use a newer C++ compiler such as that one comes with VS 2008 or VS 2010. Windows Embedded Compact 7 Platform Builder comes with good compiler that can be utilized. To use the compiler of EC7 PB, follow the instruction of Adeneo: "Building WEC7 application with ARMv7 compiler" in the following link http://www.adeneo-embedded.com/content/download/10454/138018/file/Building%20WEC7%20application%20with%20ARMv7%20compiler%20-%20TI%20Benchmark.pdf

The "/QRfpe-" option tells the compiler to suppress the calling of FPCRT functions for floating point normal operations (such as multiplications, additions....etc) and use hardware VFP instructions instead. However, transcendental function will still be linked to FPCRT.dll.

To test the software performance under different configurations, the Fast Fourier Transform (FFT) function is used with number of points of 16384. The majority of FFT calculations are floating point multiplications and additions with some calls to sine and cosine functions. The time of performing 100 FFT operation is estimated using GetTickCount() function. Table 1 below list the processing times for different configurations. Managed means implementing the calculations inside .NET entirely, whereas Unmanaged code refers to implementing the calculation inside pure native environment. Mixed code refers to using .NET application that calls native FFT function built inside a DLL.

Table-1 Windows CE 6 with S3C6410 Processing Time (sec)

Configuration	Managed	Unmanaged		Mixed	
		/QRfpe	/QRfpe-	/QRfpe	/QRfpe-
Microsoft FPCRT	105	NA	NA	NA	NA
ARM VFPv2 FPCRT	75.4	15.8	9.5	18.6	12.3

It is clear from Table-1 that using ARM FPCRT has led to some time saving over Microsoft FPCRT in the managed code application due to the using of hardware coprocessor in FPCRT.dll. However, the performance is still not optimized due to native/managed code barrier. When the same calculations are performed by pure unmanaged code, the processing time abruptly drops to 15.8 sec when compiling without VFPv2 instructions. This is reasoned to the efficient interaction between unmanaged code and FPCRT.dll functions. The time is further reduced to only 9.5 sec when the project is compiled with VFPv2 instruction support because multiplications and additions are performed directly by the VFP coprocessor.

When mixed configuration is implemented, the processing time is slightly higher than that of pure unmanaged code but far less than the time of pure managed code. This is due to the latency of P/Invoking FFT function by managed code for 100 times. However, great time saving of more than 6 times is obtained when comparing with pure managed code implementation.

3. Cortex-A8 with Windows Embedded Compact 7

In this section, an ARM Cortex-A8 processor will be tested under Windows EC7. The selected board has Samsung S5PV210 processor running at 1GHz and featuring VFPv3-D32 with NEON support. One of the enhancements made to WIN EC7 is its ability to support hardware VFP without any third-party

tools or plug-in. The Platform Builder of Windows EC7 already has the required VFP standard library which is compatible with most of the ARM processors. The created FPCRT.dll contains the hardware VFP instructions and VFP transcendental functions.

When the same scenario applied before is used to test the Cortex-A8 platform, the processing time is generally less than that in ARM11 platform due to the clock and architectural enhancements in Cortex-A8. Table-2 shows the processing times for different configurations. It should be noted that in order to support VFPv3 code generation during compilation of C++ code under Visual Studio 2008, ARMv7 instruction set must be enabled. To do so, the procedure suggested by Adeneo has been applied as follows:

1. An SDK that support ARMv7 has been generated for S5PV210 board and then installed.

2. ARM C++ compiler of WIN EC7 platform builder has been used by adding a new directory to the C++ compiler for the SDK generated above.

3. The options "`/QRarch7  /arch:VFPv3-D32 /QRfpe-`" has been added as additional compiler options for the C++ project to force the compiler generate ARMv7 instructions and also generate hardware VFP instructions for normal floating points operations.

Note:

In order to support ARMv7 project linking, two directories (listed below) must be created and the static libraries must be cloned from the nearest architecture to these directories.

$(VCInstallDir)ce\lib\lib\ARMv7

$(VCInstallDir)ce\atlmfc\lib\ARMv7

Table-2 Windows EC 7 with S5PV210 Processing Time (sec)

Configuration	Managed	Unmanaged		Mixed	
		/QRfpe	/QRfpe-	/QRfpe	/QRfpe-
VFPv3 FPCRT	55.2	9.5	4.0	12.6	5.6

It is clear from Table-2 that the processing time of A8@1GHz is less than one-half of the time of ARM11@667MHz for "`QRfpe-`" configuration despite the fact that the clock is shifted by 150% only. It can be concluded that the VFPv3 is more efficient than VFPv2. The processing time of mixed code is about 1/10 of the time of pure managed code.