EMBEDDED IMAGE PROCESSING ON THE TMS320C6000™ DSP

Examples in Code Composer Studio™ and MATLAB

EMBEDDED IMAGE PROCESSING ON THE TMS320C6000™ DSP

Examples in Code Composer Studio™ and MATLAB

Shehrzad Qureshi

Shehrzad Qureshi
Labcyte Inc., Palo Alto, CA
USA

Embedded Image Processing on the TMS320C6000™ DSP
Examples in Code Composer Studio™ and MATLAB

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

ISBN 0-387-25280-0 e-ISBN 0-387-25281-9 Printed on acid-free paper.
ISBN 978-0387-25280-3

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11055570

springeronline.com

TRADEMARKS

The following list includes commercial and intellectual trademarks belonging to holders whose products are mentioned in this book. Omissions from this list are inadvertent.

Texas Instruments, TI, Code Composer Studio, RTDX, TMS320C6000, C6000, C62x, C64x, C67x, DSP/BIOS and VelociTI are registered trademarks of Texas Instruments Incorporated.
MATLAB is a registered trademark of The MathWorks, Inc.
ActiveX, DirectDraw, DirectX, MSDN, Visual Basic, Win32, Windows, and Visual Studio are trademarks of Microsoft.
Intel, MMX, Pentium, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

DISCLAIMER

Product information contained in this book is primarily based on technical reports and documentation and publicly available information received from sources believed to be reliable. However, neither the author nor the publisher guarantees the accuracy and completeness of information published herein. Neither the publisher nor the author shall be responsible for any errors, omissions, or damages arising out of use of this information. No information provided in this book is intended to be or shall be construed to be an endorsement, certification, approval, recommendation, or rejection of any particular supplier, product, application, or service.

CD-ROM DISCLAIMER

Dedication

To my family, for instilling in me the work ethic that made this book possible, and Lubna, for her continued support.

Contents

Preface

The question might reasonably be asked when first picking up this book – why yet another image processing text when there are already so many of them? While most image processing books focus on either theory or implementation, this book is different because it is geared towards *embedded* image processing, and more specifically the development of efficient image processing algorithms running on the Texas Instruments (TI) TMS3206000™ Digital Signal Processor (DSP) platform. The reason why I wrote this book is that when I first started to search for material covering the TMS3206000 platform, I noticed that there was little focus on image processing, even though imaging is an important market base that TI targets with this particular DSP family. To be sure, there are plenty of books that explain how to implement one-dimensional signal processing algorithms, like the type found in digital communications and audio processing applications. And while I found a chapter here or there, or a lingering section that mentioned some of the techniques germane to image processing, I felt that a significant portion of the market was not being sufficiently addressed. For reasons that will hopefully become apparent as you read this book, image processing presents its own unique challenges and it is my sincere hope that you find this book helpful in your embedded image processing endeavors.

For a myriad of reasons, implementing data intensive processing routines, such as the kind typified by image processing algorithms, on embedded platforms presents numerous issues and challenges that developers who travel in the "workstation" or "desktop" realm (i.e. UNIX or Wintel platforms) typically do not need to concern themselves with. To illustrate just a few of these issues, consider the task of implementing an

efficient two-dimensional Discrete Wavelet Transform (DWT) on a DSP, a topic covered extensively in Chapter 6 of this book. Such an algorithm might be needed in a digital camera to save images in the JPEG-2000 format or in a software-based MPEG4 system – wavelet coding of images has in the past few years supplanted the discrete cosine transform as the transform of choice for state-of-the-art image compression.

The DWT, like many other similar transforms commonly encountered in signal and image processing, is a *separable* transform. This means that the two-dimensional (2D) form of the transform is computed by generalizing the one-dimensional (1D) form of it – in other words, by first performing the 1D transform along one dimension (for example, each of the individual rows in the image), and then rerunning the transform on the output of the first transform in the orthogonal direction (i.e., the columns of the transformed rows). A software implementation of a 2D separable transform is relatively straightforward, if a routine exists that performs the 1D transform on a vector of data. Such code would most likely march down each row, invoking the aforementioned routine for each row of image data, thus resulting in a matrix of row-transformed coefficients. This temporary image result could then be *transposed* (the rows become the columns and vice-versa), and the same process run again, except this time as the algorithm marches down the row dimension, the columns are now transformed. Lastly, a second transposition reshuffles the image pixels back to their original orientation, and the end result is the 2D transform. Both the 2D DWT and the 2D Fast Fourier Transform (FFT) can be computed in such a manner.

This implementation strategy has the benefit of being fairly easy to understand. The problem is that it is also terribly inefficient. The two transposition operations consume processor cycles, and moreover lead to increased memory consumption, because matrix transposition requires a scratch array. On typical desktop platforms, with their comparatively huge memory footprint and multi-gigahertz clock frequencies, who cares? On these platforms, it is oftentimes possible to get away with such first-cut implementations, and still obtain more than acceptable performance. Of course, this is not always going to be the case but the name of the game in developing efficient code is to optimize only if needed, and only where needed. Thus if the initial implementation is fast enough to meet the desired specifications, there is little to gain from making algorithmic and subsequent low-level optimizations.

The contrast is quite stark in the embedded DSP world however; embedded DSP cores increasingly find their way into real-time systems, with hard deadlines that must be met. And if the system happens to not be real-time, they are quite often memory and/or resource constrained, perhaps to keep costs and power consumption down to acceptable levels (consider,

for example, a camera cell-phone). As a consequence, the situation here is that there is typically far less leeway – the clock speeds are somewhat slower, and memory inefficiencies tend to have an amplified effect on performance. Hence memory, and especially fast on-chip memory (which comes at a premium) absolutely must be managed correctly. With respect to the 2D DWT example, the "canonical" form of the algorithm as just described can be altered so that it produces identical output, but does not require the use of a matrix transposition. Such an optimization is described in Chapter 6. As one would expect, as these types of optimizations are incorporated into an algorithm, its essential nature tends to become clouded with the details of the tricks played to coax more performance out of an implementation. But that is the price one pays for speed – clarity suffers, at least with respect to a "reference" implementation.

And it only gets more involved as the design process continues. Images of any reasonable size will not fit in on-chip RAM, which has just been identified as a rate-limiting factor in algorithm performance. Because the latencies involved with accessing off-chip RAM are so severe, an optimal 2D DWT transform should incorporate strategies to circumvent this problem. This optimization exploits spatial locality by shuttling blocks of data between internal and external RAM. For example, as individual rows or a contiguous block of an image is needed, they should be copied into internal RAM, transformed, and then sent back out to external RAM. This process would then continue to the next block, until the entire image has been transformed. A likely response from seasoned C/C++ developers would probably be to use the venerable `memcpy` function to perform these block memory copies. As it turns out, in the fully optimized case one should use Direct Memory Access (DMA) to increase the speed of the block memory copy, which has the added benefit of freeing the processor for other duties. Taking matters to the extreme, yet another optimization may very well entail interleaving the data transfer and data processing. First, one might set up the DMA transfer, and then process a block of data while the DMA transfer is occurring in the background. Then, when the background DMA transfer completes, process this new set of data while simultaneously sending the just processed block of data back out to off-chip memory, again via DMA. This procedure would continue until all image blocks have been processed.

As is evident from this description, what was once a reasonably straightforward algorithm implementation has quickly become obfuscated with a myriad of memory usage concerns. And memory is not the end of this story! Many DSPs are "fixed-point" devices, processors where floating-point calculations are to be avoided because they must be implemented in software. Developing fixed-point algorithm implementations, where any floating-point calculations are carried out using integer representations of

numbers and the decimal point is managed by the programmer, opens up a whole new can of worms. The algorithm developer must now contend with a slew of new issues, such as proper data scaling, quantization effects of filter coefficients, saturation and overflow, to name just a few. So not only has our optimized 2D DWT algorithm been tweaked so that it no longer requires matrix transpositions and is memory efficient through the usage of DMA block memory copies, but any floating-point data is now treated as bit-shifted integer numbers that must be managed by the programmer.

And even still, there are many other issues that the preceding discussion omits, for example the inclusion of assembly language, vendor-specific compiler intrinsics, and a real-time operating system, but the general idea should now be clear. Even just a few of these issues can turn a straightforward image processing algorithm into a fairly complicated implementation. Taken together and all at once, these concerns may seem overwhelming to the uninitiated. Bridging this gap, between the desktop or workstation arena and the embedded world, is this book's raison d'etre. Developers must fully understand not only the strengths and weaknesses of the underlying technology, but also the algorithms and applications to the fullest extent in order to implement them on a DSP architecture in a highly-optimized form. I come from an image processing background, having developed numerous production-quality algorithms and software spanning the gamut of environments – from embedded DSPs to dual Pentium and SGI workstations – and my primary goal is to ease the transition to the embedded DSP world, which as evidenced by this case study presents itself with a set of very unique challenges.

Another motivation for my writing this book, aside from the fact that there is not currently a book on the market covering embedded image processing, is to shed some light on the "black magic" that seems to accompany embedded DSP development. In comparison to developing non-embedded software, during the development of the software that accompanies this book I was continually stymied by one annoyance or another. Certain operations that would work with one DSP development platform seemingly would not work on another, and vice-versa. Or some simple examples from the TI online help tutorial would not work without certain modifications made to linker files, build settings, or the source code. There were so many issues that to be truthfully honest I have lost track of many of them. I hope that reading this book and using the code and projects that are on the CD-ROM will help you in that ever important, yet elusive, quest for "time to market."

Shehrzad Qureshi
shehrzad_q@hotmail.com

Acknowledgments

There are many fine people who had a hand in this work, and I would like to acknowledge their effort and support. First and foremost, my soon-to-be wife, Lubna, provided invaluable feedback with regards to the composition of the book and spent countless hours reviewing drafts of the material. Without her constant encouragement and total understanding, it is doubtful this book would have ever seen the light of day and for that, I am forever grateful. I have also been blessed to have had the pleasure of working with many brilliant engineers and scientists at various employment stops along the way, and without them to learn from I could never have developed my love of algorithms and programming. I thank Steve Ling and others at Texas Instruments for their generosity in lending DSP development boards that made this book possible, as well as Cathy Wicks for her help along the way. I sincerely appreciate the support extended to me by the folks at The MathWorks, in particular Courtney Esposito who disseminated early drafts of the manuscript for technical review. Finally, I would like to thank Springer for their continued support in this endeavor, especially Melissa Guasch, Deborah Doherty, and Alex Greene for their help in the final preparation of the manuscript.

Chapter 1

INTRODUCTION

When engineers or scientists refer to an image, they are typically speaking of an optical representation of a scene acquired using a device consisting of elements excited by some light source. When this scene is illuminated by a light source, these elements subsequently emit electrical signals that are digitized to form a set of "picture elements" or *pixels*. Together these pixels make up a *digital* image. Many of these devices are now driven by Digital Signal Processors, for reasons explained in 1.6. The imaging device may take the form of a camera, where a photographic image is created when the objects in the scene are illuminated by a natural light source such as the sun, or an artificial light source, such as a flash. Another example is an x-ray camera, in which case the "scene" is typically some portion of the human body or a dense object (e.g., luggage in an airport security system), with the light source consisting of x-ray beams. There are many other examples, some of which do not typically correspond to what most people think of as an image. For example, in the life sciences and biomedical fields there are numerous devices and instruments that can be thought of as cameras in some sense, where the acquisition detectors are photodiodes excited by some type of infrared light. This book describes image processing algorithms that operate on all sorts of images, and provides numerous implementations of such algorithms targeting the Texas Instruments (TI) TMS320C6000™ DSP platform. Prior to embarking on this journey, this first chapter introduces the structure of the book and the representation of digital images, and the second chapter provides background information on the tools used to develop image processing algorithms.

1.1 STRUCTURE AND ORGANIZATION OF THE BOOK

Because the whole thrust of this book are *efficient* implementations of image processing algorithms running on *embedded* DSP systems, it is not sufficient to simply present an algorithm and describe a first-cut implementation that merely produces the correct output. The primary goal is efficient algorithm implementations, while a secondary goal is to learn how to utilize the appropriate TI technologies that aid in the development and debugging of code that can be used in real-world applications. Achieving these goals takes time, and as such we are somewhat constrained by space. As a result, this book is *not* intended to be a complete coverage of image processing theory and first principles, for that the reader is referred to [1] or [2]. However, what you will find is that while such books may give you the mathematical and background knowledge for understanding how various image processing algorithms work in an abstract sense, the transition from theory to implementation is a jump that deserves more attention than is typically given. In particular, taking a description of an image processing algorithm and coding an efficient implementation in the C language on an embedded fixed-point and resource-constrained DSP is not for the faint of heart. Nowadays, given the proliferation of a variety of excellent "rapid-generation" high-level technical computing environments like MATLAB® (especially when coupled with the Image Processing Toolbox) and various software libraries like the Intel® Integrated Performance Primitives, it is not overtly difficult to put together a working image processing prototype in fairly short order. We will use both of the aforementioned software packages in our quest for embedded DSP image processing, but bear in mind that it is a windy road.

The meat of this book is split amongst Chapters 3-6, with some ancillary material appearing in the two appendices. This chapter contains introductory material and Chapter 2 is important background information on the various tools employed throughout the rest of the book. Chapters 3-6 roughly cover four general categories of image processing algorithms:

1. **Chapter 3**: image enhancement via spatial processing techniques (point-processing).
2. **Chapter 4**: image filtering (linear, non-linear, and adaptive).
3. **Chapter 5**: image analysis (edge-detection and segmentation).
4. **Chapter 6**: wavelets (with applications to edge detection and image enhancement).

Due to the challenging nature of embedded development, the strategy is to start off simple and then progressively delve deeper and deeper into the

intricate implementation details, with the end game always being an efficient algorithm running on the DSP. The book follows a "cookbook" style, where an image processing algorithm is first introduced in its theoretical context. While this is first and foremost a practitioner's book, it goes without saying that a solid understanding of the theoretical underpinnings of any algorithm is critical to achieving a good implementation on a DSP. After the theoretical groundwork has been laid, examples in MATLAB are used to drive home the core concepts behind the algorithm in question, without encumbering the reader with the details that inevitably will follow. Depending on the situation, it may be the case that the MATLAB code is ported to C/C++ using Visual Studio .NET 2003. These Windows applications allow for interactive visualization and prove invaluable as parallel debugging aids, helping to answer the often posed question "Why doesn't my C code work right on my DSP, when it works just fine on the PC?" And of course, because this book is primarily targeted at those who are implementing embedded image processing algorithms, each algorithm is accompanied by an implementation tested and debugged on either the C6701 Evaluation Module (EVM) or C6416 DSP Starter Kit (DSK). Both of these DSP platforms are introduced in 2.1.3. It should be noted that majority of the TI image-processing implementations that accompany this book use the C6416 DSK, as it contains a more recent DSP and TI appears to be phasing out the EVM platform.

1.2. PREREQUISITES

In order to get the most out of this book, it is expected that the reader is reasonably fluent in the C language and has had some exposure to MATLAB, C++, and the TI development environment. If the reader is completely new to embedded development on the TI DSP platform, but does have some experience using Microsoft Visual Studio or a similar integrated development environment (IDE), it should not be too difficult to pick up Code Composer Studio™ (CCStudio). CCStudio is heavily featured in this book and is TI's answer to Visual Studio. While one can always fall back to command-line compilation and makefiles, CCStudio incorporates many advanced features that make programming and debugging such a joy, as compared to the days of gdb, gcc, vi, and emacs.

One does not need to be an expert C++ programmer in order to make sense of the Visual Studio projects discussed in the book. The use of C++ is purposely avoided on the DSP, however it is the language of choice for many programmers building scientific and engineering applications on Windows and UNIX workstations. For high-performance, non-embedded

image processing, I would hasten to add that it is the only choice, perhaps leaving room for some assembly if need be. Nevertheless, I have eschewed many cutting-edge C++ features – about the most exotic C++ code one will encounter is the use of namespaces and perhaps a sprinkling of the Standard Template Library (STL). An understanding of what a class and method are, along with some C++ basics such as exception handling and the standard C++ library is all that is required, C++-wise.

The source code for all Visual Studio projects utilize Microsoft Foundation Classes (MFC) and GDI+ for their GUI components (see 2.4). For the most part, the layout of the code is structured such that one need not be a Windows programming guru in order to understand what is going on in these test applications. Those that are not interested in this aspect of development can ignore MFC and GDI+ and simply treat that portion of the software as a black box. Any in-depth reference information regarding Microsoft technologies can always be found in the Microsoft Developer's Network (MSDN)[3].

Lastly, a few words regarding mathematics, and specifically signal processing. In this book, wherever image processing theory is presented, it is just enough so that the reader is not forced to delve into an algorithm without at least the basics in hand. Mainly this is due to space constraints, as the more theory that is covered, the fewer algorithms that can fit into a book of this size. When it comes right down to it, in many respects image processing is essentially one-dimensional signal processing extended to two dimensions. In fact, it is often treated as an offshoot of one-dimensional signal processing. Unfortunately, there is not enough space to thoroughly cover the basics of one-dimensional DSP applications and so while some signal processing theory is covered, the reader will gain more from this book if they have an understanding of basic signal processing topics such as convolution, sampling theory, and digital filtering. Texts covering such one-dimensional signal processing algorithms with applications to the same TI DSP development environments utilized in this book include [4-6].

1.3 CONVENTIONS AND NOMENCLATURE

Many of the featured image processing algorithms are initially illustrated in pseudo-code form. The "language" used in the pseudo-code to describe the algorithms is not formal by any means, although it does resemble procedural languages like C/C++ or Pascal. For these pseudo-code listings, I have taken the liberty of loosely defining looping constructs and high-level assignment operators whose definitions should be self-explanatory from the

context in which they appear. The pseudo-code also assumes zero-based indexing into arrays, a la C/C++.

Any reference to variables, functions, methods, or pathnames appear in a non-proportional `Courier` font, so that they stand out from the surrounding text. In various parts of the book, code listings are given and these listings use a 10-point non-proportional font so they are highlighted from the rest of the text. This same font is used wherever any code snippets are needed. Cascading menu selections are denoted using the pipe symbol and a bold-faced font, for example **File|Save** is the "Save" option under the "File" main menu.

With any engineering discipline, there is unfortunately a propensity of acronyms, abbreviations, and terminology that may seem daunting at first. While in most cases the first instance of an acronym or abbreviation is accompanied by its full name, in lieu of a glossary the following is a list of common lingo and jargon the reader should be familiar with:

- **C6x**: refers to the TMS320C6000 family of DSPs, formally introduced in 2.1. The embedded image processing algorithms implemented in this book target this family of DSPs. C6x is short-hand for the C62x, C67x, and C64x DSPs.
- **IDE**: integrated development environment. The burgeoning popularity of IDEs can be attributed in part to the success of Microsoft Visual Studio and earlier, Borland's Turbo C and Turbo Pascal build systems. IDEs combine advanced source code editors, compilers, linkers, and debugging tools to form a complete build system. In this book we utilize three IDEs (MATLAB, CCStudio, and Visual Studio), although MATLAB is somewhat different in that it is an interpreted language that does not require compilation or a separate linking step.
- **TI**: Texas Instruments, the makers of the C6x DSPs.
- **CCStudio**: abbreviation for the Code Composer Studio IDE, TI's flagship development environment for their DSP products.
- **M-file**: a collection of MATLAB functions, analogous to a C/C++ module or source file.
- **MEX-file**: MATLAB callable C/C++ and FORTRAN programs. The use and development of MEX-files written in C/C++ is discussed in Appendix A.
- **toolbox**: add-on, application-specific solutions for MATLAB that contain a family of related M-files and possibly MEX-files. In this book, the Image Processing Toolbox, Wavelet Toolbox, and Link for Code Composer Studio are all used. For further information on MATLAB toolboxes, see [7].

- **host**: refers to the PC where CCStudio is running. The host PC is connected to a TI DSP development board via USB, PCI, or parallel port.
- **EVM**: an abbreviation for evaluation module, introduced in 2.1.3. The EVM is a PCI board with a TI DSP and associated peripherals used to develop DSP applications. All EVM code in this book was tested and debugged on an EVM containing a single C6701 DSP, and this product is referred to in the text as the C6701 EVM.
- **DSK**: refers to a "DSP starter kit", also introduced in 2.1.3. The DSK is an external board with a TI DSP and associated peripherals, connected to a host PC either via USB or parallel port (see Figure 2-4). All DSK code in this book was tested and debugged on a DSK with a C6416 DSP, and this product is referred to in the text as the C6416 DSK.
- **target**: refers to the DSP development board, either an EVM or DSK.

Finally, a few words regarding the references to TI documentation – the amount of TI documentation is literally enormous, and unfortunately it is currently not located in its entirety in a central repository akin to MSDN. Each TI document has an associated "literature number", which accompanies each reference in this book. The literature number is prefixed with a four-letter acronym – either SPRU for a user manual or SPRA for an application report. Some of these PDFs are included with the stock CCStudio install, but all of them can be downloaded from www.ti.com. For example, a reference to SPRU653.pdf (user manual 653) can be downloaded by entering "SPRU653" in the keyword search field on the TI web-site, if it is not already found within the `docs\pdf` subdirectory underneath the TI install directory (typically `C:\TI`).

1.4 CD-ROM

All the source code and project files described in Chapters 3-6 and the two appendices are included on the accompanying CD-ROM. Furthermore, most of the raw data – images and in one case, video – can also be found on the CD-ROM. The CD is organized according to chapter, and the top-level `README.txt` file and chapter-specific `README.txt` files describe their contents more thoroughly.

The host PC used to build, test, and debug the DSP projects included on the CD-ROM had two versions of CCStudio installed, one for the C6701 EVM and another for the C6416 DSK. As a result, chances are that the projects will not build on your machine without modifications made to the

CCStudio project files. The default install directory for CCStudio is `C:\TI`, and as the DSP projects reference numerous include files and TI libraries, you will most likely need to point CCStudio to the correct directories on your machine (the actual filenames for the static libraries and header files should remain the same). There are two ways of going about this. One way is to copy the project directory onto the local hard drive, and then open the project in CCStudio. CCStudio will then complain that it cannot find certain entities referenced in the `.pjt` (CCStudio project) file. CCStudio then prompts for the location of the various files referenced within the project which it is unable to locate. This procedure is tedious and time-consuming and an alternate means of accomplishing the same thing is to directly edit the `.pjt` file, which is nothing more than an ASCII text file. Listing 1-1 shows the contents of the `contrast_stretch.pjt` CCStudio (version 2.20) project file, as included on the CD-ROM. The lines in bold are the ones that need to be tailored according to your specific installation.

Listing 1-1: The contents of an example CCStudio project file from Chapter 3, `contrast_stretch.pjt`. If the default options are chosen during CCStudio installation, the bold-faced directories should be changed to `C:\TI` – otherwise change them to whatever happens to be on your build machine.

```
CPUFamily=TMS320C67XX
Tool="Compiler"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="C:\TIC6701EVM\c6000\cgtools\lib\rts6701.lib"
Source="C:\TIC6701EVM\c6200\imglib\lib\img62x.lib"
Source="C:\TIC6701EVM\myprojects\evm6x\lib\evm6x.lib"
Source="C:\TIC6701EVM\myprojects\evmc67_lib\Dsp\Lib\devlib\De
v6x.lib"
Source="C:\TIC6701EVM\myprojects\evmc67_lib\Dsp\Lib\drivers\D
rv6X.lib"
Source="contrast_stretch.c"
Source="contrast_stretch.cmd"

["Compiler" Settings: "Debug"]
Options=-g -q -fr".\Debug"
-i "C:\TIC6701EVM\myprojects\evm6x\dsp\include"
```

```
-i "C:\TIC6701EVM\c6200\imglib\include" -d"_DEBUG" -mv6700

["Compiler" Settings: "Release"]
Options=-q -o3 -fr".\Release"
-i "C:\TIC6701EVM\myprojects\evm6x\dsp\include"
-i "C:\TIC6701EVM\c6200\imglib\include" -mv6700

["DspBiosBuilder" Settings: "Debug"]
Options=-v6x

["DspBiosBuilder" Settings: "Release"]
Options=-v6x

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\contrast_stretch.map"
-o".\Debug\contrast_stretch.out" -w -x

["Linker" Settings: "Release"]
Options=-q -c -m".“Release“contrast_stretch.map"
-o".\Release\contrast_stretch.out" -w -x
```

Upon opening one of these projects, if CCStudio displays an error dialog with the message "Build Tools not installed", verify that the project directory is valid. If that does not correct the issue, then the problem is most likely the `CPUFamily` setting at the beginning of the `.pjt` file. Chances are that there is a mismatch between the current CCStudio DSP configuration and whatever is defined in the project file. This setting can be changed to whatever is appropriate (i.e. `TMS320C64XX`), although this modification is fraught with peril. At this point, it would be best to re-create the project from scratch, using the provided project file as a template. See [8] for more details on CCStudio project management and creation.

Some of the Visual Studio .NET 2003 projects on the CD-ROM may also need to be modified in a similar fashion, depending on where the Intel Integrated Performance Primitives Library and CCStudio have been installed. In this case, direct modification of the `.vcproj` (Visual Studio project) file is not recommended, as the format of these text files is not as simple as `.pjt` files. Instead, modify the include and library path specifications from within the project properties dialog within Visual Studio, via the **Project|Properties** menu selection.

1.5 THE REPRESENTATION OF DIGITAL IMAGES

Except for a single example in 3.4, this book deals entirely with digital *monochrome* (black-and-white) images, oftentimes referred to as "intensity" images, or "gray-scale" images. A monochrome digital image can be thought of as a discretized two-dimensional function, where each point represents the light intensity at a particular spatial coordinate. These spatial coordinates are usually represented in a Cartesian system as a pair of positive integer values, typically denotes in this book as *(i,j)* or *(x,y)*. The spatial coordinate system favored in this book is one where the first integer i or x is the row position and the second integer j or y is the column position, and the origin is the upper left corner of the image. Taken together, the discrete image function *f(i,j)* returns the pixel at the i^{th} row and j^{th} column. Depending on the language, the tuples *(i,j)* or *(x,y)* may be zero-based (C/C++) or one-based (MATLAB) language. A digital image is usually represented as a *matrix* of values, and for example in MATLAB you have

$$f(i,j)=\begin{bmatrix} f(1,1) & f(1,2) & f(1,3) & \cdots & f(1,N) \\ f(2,1) & f(2,2) & f(2,3) & \cdots & f(2,N) \\ \vdots & \vdots & \vdots & & \vdots \\ f(M,1) & f(M,2) & f(M,3) & \cdots & f(M,N) \end{bmatrix}$$

The number of rows in the above image is M and the number of columns is N – these variable names show up repeatedly throughout the MATLAB code in this book. In the C code, the preprocessor symbols `X_SIZE` and `Y_SIZE` refer to the number of rows and number of columns, respectively.

In the C and C++ languages, we represent an image as an array of two dimensions, and since both C and C++ use zero-based indices and brackets to specify a pointer dereference, the C/C++ equivalent to the above image matrix is

$$f(i,j)=\begin{bmatrix} f[0][0] & f[0][1] & f[0][2] & \cdots & f[0][N-1] \\ f[1][0] & f[1][1] & f[1][2] & \cdots & f[1][N-1] \\ \vdots & \vdots & \vdots & & \vdots \\ f[M-1][0] & f[M-1][1] & f[M-1][2] & \cdots & f[M-1][N-1] \end{bmatrix}$$

As explained in 3.2.2, for performance reasons we usually do not store images in the above fashion when coding image processing algorithms in

C/C++. Very often, the image matrix is "flattened" and stored as a one-dimensional array. There are two prevalent ordering schemes for storing flattened 2D matrices, *row-major* and *column-major*. In row-major ordering, the matrix is stored as an array of rows, and this is the format used in C/C++ when defining 2D arrays. In the column-major format, which MATLAB and FORTRAN use, the matrix is ordered as an array of columns. Table 1-1 illustrates both of these formats, for the above example image matrix.

Table 1-1. Flattening an image matrix (with M columns and N rows) into a one-dimensional array.

row-major	column-major
$f[0][0]$	$f[0][0]$
$f[0][1]$	$f[1][0]$
$f[0][2]$	$f[2][0]$
...	...
$f[0][N-1]$	$f[M-1][0]$
$f[1][0]$	$f[0][1]$
$f[1][1]$	$f[1][1]$
$f[1][2]$	$f[2][1]$
...	...
$f[1][N-1]$	$f[M-1][1]$
...	...
$f[M-1][N-1]$	$f[M-1][N-1]$

The individual pixels in digital monochrome images take on a finite range of values. Typically, the pixel values $f(i,j)$ are such that

$$0 \leq f(i,j) < 2^{bpp}$$

where "bpp" is bits-per-pixel. An individual pixel value $f(i,j)$ goes by many commonly used names including: "gray-level intensity", "pixel intensity", or sometimes simply "pixel". In this book, we largely deal with monochrome 8 bpp images, with pixel intensities ranging in value from 0 to 255. These types of images are sometimes referred to as 8-bit images.

1.6 DSP CHIPS AND IMAGE PROCESSING

For the most part, image processing algorithms are characterized by repetitively performing the same operation on a group of pixels. For example, some common arithmetic image operations involve a single image and a single scalar, as in the case of a scalar multiply:

$$g(x,y) = \alpha f(x,y)$$

The above expression translates in code to multiplying each pixel in the image f by the constant α. Bilinear operations are pixel-wise operations that assume images are of the same size, for example:

- $g(x,y) = f_1(x,y) + f_2(x,y)$
- $g(x,y) = f_1(x,y) - f_2(x,y)$
- $g(x,y) = f_1(x,y) * f_2(x,y)$
- $g(x,y) = f_1(x,y) / f_2(x,y)$

Another very common category of image processing algorithms are mask or filter operations, where each pixel $f(x,y)$ is replaced by some function of $f(x,y)$'s neighboring pixels. This digital filtering operation is of critical importance, and this class of algorithms is introduced in Chapter 4. Filtering a signal, either one-dimensional or multi-dimensional, involves repeated *multiply-accumulate*, or MAC, operations. The MAC operation takes three inputs, yields a single output, and is described as

$$\text{MAC} = (a \times b) + c$$

As described in 2.1.2, in fixed-point architectures a, b, and c are integer values whereas with floating-point architectures those three are either single-precision or double-precision quantities. The MAC operation is of such importance that is has even yielded its own benchmark, the M-MAC, or million of multiply-accumulate operations per second (other benchmarks include MIPS, or million of instructions per second and MFLOPS, or million of floating-point instructions per second). Digitally filtering an image involves repeatedly performing the MAC operation on each pixel while sliding a mask across the image, as described in Chapter 4.

All of the above operations have one overriding characteristic that stands out among all others – they involve repetitive numerical computations requiring a high memory bandwidth. In short, image processing is both very compute- *and* data-intensive. In addition, image processing applications are increasingly finding themselves in embedded systems, oftentimes in settings where real-time deadlines must be met. With respect to embedded systems, consider the digital camera or camera cell phone. Such a device requires a computing brain that performs the numerical tasks described above highly efficiently, while at the same time minimizing power, memory use, and in the case of high-volume products, cost. Add to these real-time constraints, like the type one may encounter in surveillance systems or medical devices such as ultrasound or computer-aided surgery, and all of a sudden you now

find yourself in a setting where it is very likely that a general purpose processor (GPP) is not the appropriate choice. GPPs are designed to perform a diverse range of computing tasks (many of them not numerically oriented) and typically run heavy-weight operating systems definitely not suited for embedded and especially real-time systems.

Digital Signal Processors arrived on the scene in the early 1980s to address the need to process continuous data streams in real-time. Initially they were largely used in 1D signal processing applications like various telecommunication and audio applications, and today this largely remains the case. However, the rise of multimedia in the 1990s coincided with an increasing need to process images and video data streams, quite often in settings where a GPP was not going to be used. There has been a clear divergence in the evolution of DSPs and GPPs, although every manufacturer of GPPs, from Intel to SUN to AMD, has introduced DSP extensions to their processors. But the fact remains that by and large, DSP applications differ from their GPP counterparts in that they are most always characterized by relatively small programs (especially when compared to behemoths like databases, your typical web browser, or word processor) that entail intensive arithmetic processing, in particular the MAC operation that forms the building block of many a DSP algorithm. There is typically less logic involved in DSP programs, where logic refers to branching and control instructions. Rather, what you find is that DSP applications are dominated by tightly coded critical loops. DSPs are architected such that they maximize the performance of these critical loops, sometimes to the detriment of other more logic-oriented computing tasks. A DSP is thus the best choice for high-performance image processing, where the algorithms largely consist of repetitive numerical computations operating on pixels or groups of pixels, and where such processing must take place in a low-cost, low-power, embedded, and possibly real-time, system.

There are further unique architectural characteristics of DSPs that give them an edge in signal and image processing algorithms, including zero overhead loops, specialized I/O support, unique memory structures characterized by multiple memory banks and buses, saturated arithmetic, and others. In particular, there are certain vector operations that enable a huge boost in computational horsepower. These so-called SIMD (single instruction, multiple data) instructions, designed to exploit instruction level parallelism, crop up throughout this book and are covered in Appendix B. The C64x DSP, discussed in the next chapter, is particularly well suited for image processing applications as it is a high-speed DSP with numerous instructions that map very well to the efficient manipulation of 8-bit or 16-bit pixels. For a more thorough discussion of DSP architectures and the history of the evolution of DSPs, the reader is referred to [9].

1.7 USEFUL INTERNET RESOURCES

There are a number of very useful web-sites pertaining to C6x development, and will leave the appropriate Internet search as an exercise for the reader. That being said, there is one resource above all others that anyone serious about C6x development should be aware of, and that is the Yahoo! group "c6x" (groups.yahoo.com/group/c6x/). This discussion forum is very active, and there are a few expert individuals who actively monitor the postings and are always giving their expert advice on a wide variety of topics pertaining to C6x DSP development.

In addition, the following USENET newsgroups are also important resources that should never be overlooked during an Internet search:

- **comp.dsp**: discussions on signal processing and some image processing applications, as well as general DSP development.
- **comp.soft-sys.matlab**: MATLAB-related programming.
- **sci.image.processing**: image processing algorithms.

The MathWorks also maintains the MATLAB Central File Exchange (www.mathworks.com/matlabcentral/fileexchange/), a very handy repository of user-contributed M-files. This site should be your first stop when searching for a particular MATLAB-based algorithm implementation. Finally, for Windows-related programming issues the following web-sites are highly recommended: www.codeproject.com and www.codeguru.com.

REFERENCES

1. Gonzalez, R., and Woods, R., *Digital Image Processing* (Addison-Wesley, 1992).
2. Russ, J., *The Image Processing Handbook* (CRC Press, 1999).
3. Microsoft Developer Network, http://www.microsoft.msdn.com
4. Chassaing, R., *DSP Applications Using C and the TMS320C6x DSK*, (Wiley, 2002).
5. Dahnoun, N., *Digital Signal Processing Implementation using the TMS320C6000 DSP Platform* (Prentice-Hall, 2000).
6. Tretter, S., *Communication System Design Using DSP Algorithms, With Laboratory Experiments for the TMS320C6701 and TMS320C6711* (Kluwer Academic/Plenum, 2003).
7. http://www.mathworks.com/products/product_listing/index.html?alphadesc
8. Texas Instruments, *Code Composer Studio Getting Started Guide* (SPRU509C).
9. Lapsley, P., Bier, J., Shoham, A., Lee, E., *DSP Processor Fundamentals* (Wiley, 1996).

Chapter 2

TOOLS

Even though this book has a narrow focus, it calls for a wide array of tools, some hardware (DSP development boards) but mostly software. It is the author's strong belief that the development of embedded algorithms should proceed from a high-level vantage point down to the low-level environment, in a series of distinct, clearly defined milestones. The risk in jumping headlong into the embedded environment is getting bogged down in countless details and the inevitable unforeseen engineering challenges that may or may not be directly related to the algorithm. Embedded development is *hard*, and some may claim much harder than coding a web application, GUI application, Java program, or most other software intended to run on a desktop machine. Embedded developers are much closer to the hardware, and usually have fewer computing resources available at their disposal. The saving grace is that the programs typically running on an embedded DSP system are of a much smaller footprint than a desktop or server application. Nevertheless, when you are that much closer to the hardware there are many, many issues that must be taken into account and hence the development strategy put forth in this book – start with the background, prototype whatever operation needs to be implemented, and slowly but surely work your way down to the DSP.

Although this description of embedded image processing development may appear to be characterized by fire and brimstone, in reality the overall situation has gotten much better over the years. Ease of development is proportional to the quality of the tools at your disposal, and in this chapter all of the tools encountered in this book are formally introduced. These include:

- The TMS320C6000 line of DSPs, in particular the C6416 DSP Starter Kit (DSK) and the C6701 Evaluation Module (EVM).
- MATLAB and the various toolboxes used in Chapters 3-6 to prototype image processing algorithms.
- Visual Studio .NET 2003, and the libraries used to build image processing applications that run on the various flavors of Microsoft Windows.

A small amount of background information on the TI line of DSPs is appropriate before jumping into the fray. Thus we first take a slight detour into the land of computer architecture and computer arithmetic, before getting down to business and describing the contents of our tool chest we use throughout the rest of this book.

2.1. THE TMS320C6000 LINE OF DSPS

In 1997, Texas Instruments introduced the C6x generation of DSPs. These chips were unique in that they were the first DSPs to embrace the Very Long Instruction Word (VLIW) architecture[1]. This architectural aspect of the DSP, deemed VelociTI™, enabled a degree of parallelism previously unheard of in processors of its class and is the key to their high performance. The first members of the C6x family were the fixed-point C62x (C6201, C6211, etc.) and floating-point 67x (C6701, C6711, C6713, etc.) series. These DSPs feature eight functional units (two multipliers and six arithmetic logic units, or ALUs) and are capable of executing up to eight 32-bit instructions per cycle. The C62x/C67x was followed in 2001 by the C64x series of fixed-point DSPs, which represented a large step up in processor speeds (scalable up to 1.1 GHz versus approximately 250-300 MHz for the C62x/C67x at the time of this writing) and also introduced extensions to VelociTI that have important ramifications on high-performance image processing. Figure 1-1 shows block diagrams of both architectures, illustrating their common roots.

What Figure 1-1 does not show are all of the associated peripherals and interconnect structures that are equally important to understanding the C6000 architecture. This relationship is shown in Figure 1-2 for the case of the C62x/C67x DSP. Figure 1-2 shows a processor core surrounded by a variety of peripherals and banks of memory with data shuttling across separate program and data buses. The C6000 has a relatively simple memory architecture, with a flat, byte-addressable 4 GB address space, split into smaller sections mapped to different types of RAM (SDRAM or SBSRAM). As shown in both Figures 1-1 and 1-2, there are actually two data paths in

the processor core, each containing four functional units. In the C62x/C67x, each data path has 16 32-bit registers, while in the C64x this register file is augmented with an additional 16 32-bit registers per data path. In addition, the C64x features an enhanced multiplier that doubles the 16-bit multiply rate of the C62x/C67x[2]. A full description of the entire C6x architecture is covered in [1-5], so in this section we instead focus on explaining the rationale and motivation behind VLIW and how it fits into the C6000 architecture, before moving to an important discussion on the difference between fixed-point and floating-point architectures. We conclude the section by introducing the two TI development environments used in this book.

2.1.1 VLIW and VelociTI

In modern Complex Instruction Set Computer (CISC) processors and to a lesser extent, Reduce Instruction Set Computer (RISC) processors, there is an incredible amount of hardware complexity designed to exploit instruction level parallelism (ILP). With sincere apologies for the multitude of acronyms (at least you were warned!), the general concept is that deeply pipelined and superscalar GPPs, with their branch prediction and out-of-order execution, perform significant analysis on the instruction stream at run-time, and then transform this instruction stream wherever possible to keep the processor's pipeline fully utilized. When the processor's pipeline is full, it completes an instruction with every clock cycle. As one might imagine, this analysis is an extraordinarily difficult task, and while the compiler does do some things to alleviate the burden on the CPU, the CPU is very much in the loop with regards to optimizing the flow of instructions through the pipeline.

With VLIW on the other hand, the onus is completely on the compiler, and the processor relies on the compiler to provide it with a group of instructions that are guaranteed to not have any dependencies among them. Hence, the compiler, rather than the hardware, is the driving element behind taking advantage of any ILP. The VLIW concept is an outgrowth of vector processors, like the Cray supercomputers from the 1970s, which were based on this idea of the exact same operation being performed on an array of data. We can illustrate the rationale behind this concept, by considering at a very high-level the steps any processor must take to execute an instruction stream:

1. Fetch the next instruction.
2. Decode this instruction.
3. Execute this instruction.