Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 49051, 14 pages
doi:10.1155/2007/49051

Research Article

PfeLib—A Performance Primitives Library for Embedded Vision

Christian Zinner, Wilfried Kubinger, and Richard Isaacs

Austrian Research Centers GmbH-ARC, Donau-City-Street 1, 1220 Vienna, Austria

Received 30 April 2006; Revised 1 September 2006; Accepted 14 September 2006

Recommended by Moshe Ben-Ezra

This paper presents our work on PfeLib—a high performance software library for image processing and computer vision algorithms for an embedded system. The main target platform for PfeLib is the TMS320C6000 series of digital signal processors (DSPs) from Texas instruments. PfeLib contains several new approaches for problems that are typical when developing software for embedded systems. We propose a method for image data transfer from a development host (PC) to an embedded system for test and verification. This enables step-by-step performance optimizations directly on the target platform. An optimization procedure is described that illustrates our approach for obtaining the best possible DSP performance with a reasonable development effort. Speedup improvement factors of up to 16 were achieved. Also, the problem of the limited on-chip memory on DSPs is addressed by a novel double buffering method using direct memory access (DMA), called resource optimized slicing (ROS-DMA). ROS-DMA is intended to be used instead of L2 cache and it is a core component of PfeLib—it achieves up to six times faster image processing as compared to using L2 cache.

1. INTRODUCTION

The field of computer vision (CV) offers a great variety of useful applications. It is very important for the technical feasibility and for the market acceptance to realize such applications on embedded systems like digital signal processors (DSP). DSPs have very low power consumption which enables small, portable, and even battery-driven devices with no cooling fans. On the other hand a developer of a computer vision application will face several additional difficulties when using a DSP as the target platform instead of a desktop personal computer. These difficulties arise from architectural limitations of DSPs, for example, lack of operating system services, smaller memories, lower CPU clock frequencies, fixed point architectures (no floating point units). Another issue is that testing and debugging generally require more time and effort when doing cross-platform development. And finally there are very few performance-optimized imaging libraries available that are suitable for a specific embedded platform.

Our research group is engaged in developing innovative computer vision systems in the automotive [1, 2] and intelligent vehicles area [3, 4]. Our experience has shown that new CV algorithms often require special image processing functions as their basis which cannot be found in any currently available imaging libraries.

Another motivation for creating PfeLib was that it was soon clear that there exists an enormous potential for performance optimization on the chosen DSP platform, which is the TMS320C6000 DSP from Texas Instruments (TI) [5], when beginning with an ordinary ANSI C code implementation.

This leads to several main requirements for PfeLib that can be enumerated as follows.

(i) Optimal performance shall be achieved on the target platform despite the typical limitations of embedded systems (memory, processor clock frequency, fixed point architecture, ...).

(ii) Beside of the platform-specific implementation, an additional generic ANSI C representation of the library functions shall provide both a unique definition of the functional behavior as well as the possibility to compile PfeLib on any other platform with the same functionality, although not performance optimized.

(iii) The library framework shall be an open architecture that enables a programmer who uses PfeLib to extend it with additional image processing functions.
(iv) The *Intel Performance Primitives for Image Processing* (ippIP) [6] shall serve as a benchmark for PfeLib in terms of performance and partially in terms of functionality. The ippIP are assembler optimized routines for image processing on a PC under Windows or Linux.
(v) The environment for development, test, and verification of PfeLib functions on an embedded system shall be reliable and easy to use.

The rest of this paper is organized as follows. Section 2 describes in brief the target platform that was mainly in use during development of PfeLib. Some properties are discussed about the CPU architecture as well as about the memory system. Section 3 introduces a tool for image data transfer to and from the embedded system called PfeRtdxHost. Section 4 gives an overview of the library framework of PfeLib. This section clarifies how several modules of PfeLib work together and how a computer vision application uses them. The task of adding a new algorithm to PfeLib is also described there. Section 5 deals with some performance optimization strategies applied on PfeLib functions. Two algorithms are selected and optimized step by step. Section 6 presents the method for double buffering with ROS-DMA, which is a vital component of PfeLib. Section 7 contains various diagrams with results of performance tests. The behavior on the DSP platform under different memory configurations is analyzed. The section concludes with a cross-platform comparison between PfeLib on a DSP and ippIP on a PC. Section 8 gives a final conclusion and points out some topics for future work on PfeLib.

2. TARGET PLATFORM AND DEVELOPMENT ENVIRONMENT

Although PfeLib is intended to provide equivalent APIs among various platforms (either embedded ones or not), the first hardware platform that is considered to be a target for the PfeLib to run on is the TMS320C6000 series of DSPs from Texas Instruments.

2.1. Related literature

Some recommended reading will be referenced here to provide additional insight to the primary processor platform used in this paper.

Selecting the optimal hardware platform for a CV application is of great importance. A practical approach to processor selection is given by [7], where expected production volumes and several other requirements are taken into account. The same paper gives examples for low-level computer vision for media chips similar to the main target platform of PfeLib. Software optimization methods are also discussed.

The work [8] deals with embedded image processing on the TMS320C6000 platform. After a description of the development environment and tools, several examples of image processing techniques are implemented in several ways: in a MATLAB environment, as a Visual Studio .NET application and finally on a C6416 DSP. Therefore, the differences between a PC platform and an embedded system in the image processing domain are pointed out.

Another work [9] has a similar approach, with an even more detailed introduction into the same DSP platform, but it deals with digital signal processing in general, not focused on image processing.

2.2. Target platform architecture

Two processor models from the TMS320C6000 series were especially chosen for specific performance optimizations:

(i) C6416 fixed point DSP: up to 1GHz, 8000MIPS (2005),
(ii) C6713 floating point DSP: up to 300 MHz, 1350 MFLOPS (2005).

The C6416 is a fixed point processor. Floating point operations must either be emulated by software or the algorithm must be designed to perform all time critical operations in the integer or fixed point domain. Existing algorithms must be ported to the fixed point data domain, which usually involves a loss of computation precision.

Common to all members of the C6000 DSP platform is that they have a *very long instruction word* (VLIW) architecture. There are eight functional units in the CPU that can potentially operate in parallel. One instruction for each of the eight units can be packed into a long instruction word. This means that on a VLIW processor the parallelism of execution is already defined at compile time. It has a great influence on the achievable performance whether the compiler is able to employ all parallel units to a maximum extent.

A very important key technique to optimize loops on VLIW machines is software pipelining. The work [10] is an early introduction into software pipelining, and [11] dives into its more theoretical backgrounds. Software pipelined loops can be coded by hand in assembler, eventually supported by dedicated tools as described in [12]. Texas Instruments' optimizing C/C++ compiler [13] is also capable of this type of optimization. If one wants to write high-performing programs in C, it is also important to know about the basic principles of software pipelining.

Figure 1 illustrates the basic concept of software pipelining. We consider a loop body that consists of five stages called *A,B,C,D,E* that must be performed one after the other. The loop body must be executed n times. We assume that the VLIW processor is equipped with enough functional units to work on all these stages in parallel. But this is not possible, because each stage needs the result of its predecessor as input data. What can be done is to process different stages of different loop iterations in parallel. This happens in the pipelined kernel of the loop. Once the loop kernel is reached, a whole iteration of the loop can be finished in the same time that otherwise was needed for only one stage of a loop body. As a prerequisite there must be a *pipelined-loop prolog* that initializes the first few loop iterations. A similar procedure occurs at the end of the loop when the *pipelined-loop epilog* finishes the

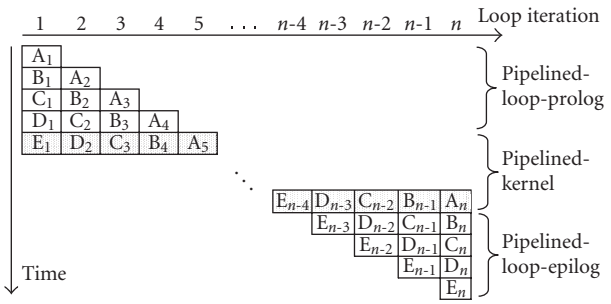


Figure 1: Software pipelined loop.

remaining stages of the last loop iterations. In the example of Figure 1, a speedup of factor 5 can be achieved compared to a nonpipelined version. But it must be taken into account that the code size also increases.[1]

As image processing is very data intensive, the memory system of a target platform has great influence on the achievable execution speed. The C6000 DSPs have a hierarchical memory architecture with several different memory types which trade off between speed and size. Only the L1 program and data caches and the internal L2 SRAM (or IRAM) are integrated on-chip. Table 1 shows the relationship between access speed and size of various types of memory. The latency times are given in CPU cycles. Up to one quarter of the IRAM can be configured to serve as L2 cache for the external memory. The best performance can only be achieved when both code and data reside in the IRAM.

2.3. Development environment

TI provides an integrated development environment (IDE) for the C6000 platform called Code Composer Studio (CCS). The IDE runs under Windows. Various simulators for the target devices are available. They are also driven through the CCS, so it gives the same "look and feel" to test a program on the simulator as well as on real target hardware.

In addition to using the simulators extensively during the development of PfeLib, two different *DSP Starter Kits (DSK)* from the company *Spectrum Digital* [15] are in use for development and testing on real hardware. One test system is a TMS320C6713 DSK which uses the C6713 processor with 225 MHz clock frequency and 8 MB SDRAM. The second test system is a TMS320C6416 DSK with a C6416 DSP clocked at 1 GHz and 16 MB SDRAM. These boards are connected to the development host via a USB/JTAG interface.

3. DATA EXCHANGE HOST—DSP

A common method of software development on DSPs is implementing the programs on a PC platform and afterwards porting the software to the embedded system. This approach has many disadvantages because platform-specific optimization techniques cannot be employed from the beginning of the development effort. The reason for using this method lies often in the absence of a reasonable test and verification environment for the embedded system.

We wanted to have the ability to start creating high performance image processing routines directly on the target platform. A system for transferring image data from a host computer to the DSP-target and vice versa is an essential precondition. Creating images for input test data and verifying the processed images can then be performed on the development host. A procedure for measuring the processing time for an operation completes the test environment. The desired procedure is illustrated in Figure 2. For obtaining reliable time stamps on the DSP, one of the built-in timers of the DSP is used that enables an accurate clocking of the processing time.

We programmed a so-called RTDX host client that is able to realize the workflow of Figure 2. TI's RTDX (real-time data exchange) [16] is a method for transferring data between a signal processor and a development host. An RTDX connection consists of a small software component on the DSP target and a host-application on the PC. These components communicate with each other via the JTAG interface and the Code Composer Studio. The latter operates as a COM server for the host client. No extra peripheral component is necessary on the target side.

PfeRtdxHost is an application with a graphical user interface (GUI) that runs under Windows. PfeRtdxHost is based on the open source software library *CxImage* [17]. This application provides a lot of functionality that is common to image manipulation programs and that is quite useful for creating, saving, loading, manipulating, or viewing test images. Hence, we could create a powerful development environment with limited effort. Additional useful properties of PfeRtdxHost are as follows.

(i) It works in the same manner for both the real DSP hardware as well as the DSP simulator without modifications of the target program.

(ii) It supports both standard- and high-speed-RTDX. HSRTDX is available with *XDS560*-class [18] JTAG emulation controllers and it offers a high data-rate of 2 MB/s. HSRTDX is indeed real-time capable, because there are no software-breakpoints necessary for the data transfer. Direct memory access (DMA)-based, nonblocking data-transfers are possible while the target application executes.

(iii) It supports both 16-bit and 32-bit image-data (signed/unsigned) with special viewing modes for efficient visual inspection and reliable functional verification.

(iv) It has a differential-viewing-mode that shows only the differences between a processed and a reference image. This proved to be very useful during the conversion of algorithms from floating point to fixed point arithmetic, where sometimes deviations in the range of a few LSBs have to be taken into account.

Figure 3 shows a screenshot of PfeRtdxHost that demonstrates an effective use-case of the differential image view. The upper left image is connected to an RTDX input channel

[1] The newest C64x+ core introduces a so-called SPLOOP buffer that can avoid this increase in code size [14].

Table 1: Memory hierarchy overview for the used DSP platforms.

Platform	L1D cache		Internal L2 SRAM		External memory (SDRAM)	
	Capacity	Latency	Capacity	Latency	Capacity	Latency
C6713	4 Kbyte	0 Cycles	256 Kbyte	4 Cycles	up to 1 Gbyte	Rd: ∼45/Wr: ∼20 Cycles
C6416	16 Kbyte	0 Cycles	1024 Kbyte	6 Cycles	up to 1 Gbyte	Rd: ∼110/Wr: ∼40 Cycles

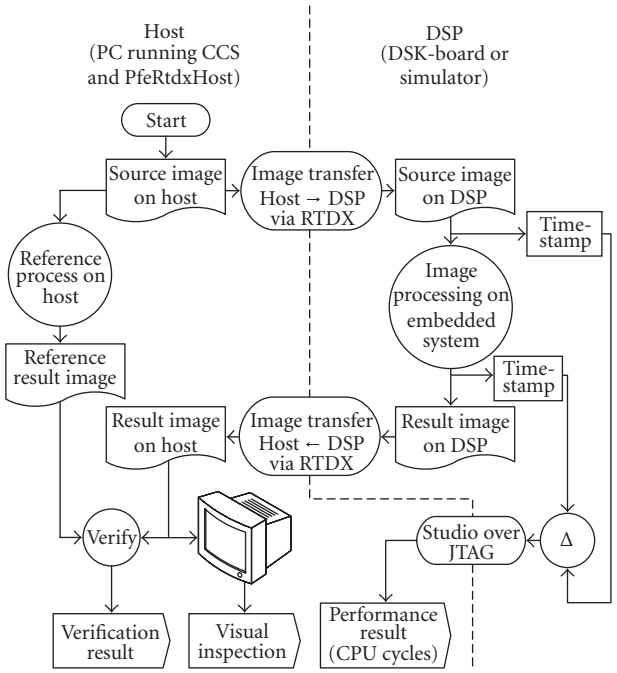


Figure 2: Desired host-to-DSP image transfer and verification process.

Figure 3: Screenshot of PfeRtdxHost.

and serves as input data for a PfeLib function running on the DSP. The operation performed on the DSP is the PfeLib function `PfeWarpAffineBack_8u_C1R()` with certain values for the warping coefficients. The affine warping is performed on the DSP and the result is sent back to PfeRtdxHost. It is visualized in the lower right image. The functionality of this image processing function is similar to the ippIP function `ippiWarpAffineBack_8u_C1R()` with linear interpolation [6], but with the difference that the internal coordinate transformations are calculated with fixed point arithmetic instead of floating point operations. This enables much faster processing on the C64x DSP—on the other hand it introduces rounding errors. Therefore, the lower left image in Figure 3 shows a reference result of the warping operation created with the ippIP function. The upper right image is configured to receive the result of the PfeLib operation and its viewing mode is set to display the difference to the reference image. The difference image naturally has signed pixel values and thus it is displayed in a color-coded mode where positive values are represented by shades of blue and negative pixels are colored in red tones. It can be seen that ∼50 percent of the differential image's pixels have a value of −1 or less and are therefore red. This clearly indicates that something is not correct with the fixed point implementation of that algorithm. And indeed, an implementation error could be found and fixed. Figure 4 shows the differential image with the correct implementation: most pixels are black and are therefore identical to the reference image, and the remaining pixels have a value of either +1 or −1, which are the small deviations that can hardly be avoided when moving from a floating point to a fixed point implementation.

4. LIBRARY FRAMEWORK

The requirements listed in Section 1 lead us, in addition to some other considerations, to a structure of source code modules, library components, and a test environment that make up the library framework of PfeLib. Figure 5 shows a typical workflow of developing an embedded CV application using PfeLib and its associated tools. The figure gives an

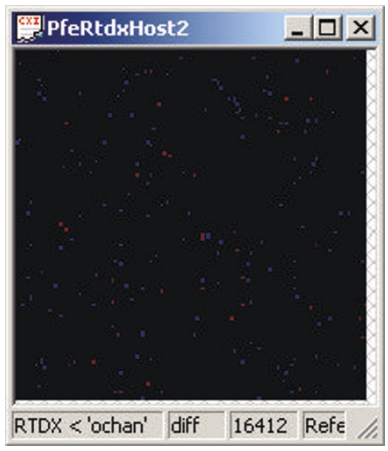


Figure 4: Differential image of the correct fixed point implementation-colored dots mark the remaining differences.

overview of the major components involved and the most important relationships between them. In this section, the components depicted in Figure 5 are referenced with Arabic numbers enclosed in boxes like [1], and the diagramed relations are referenced with uppercase letters, for example, Ⓐ. A number of simplifications were made in this illustration; any source code-like constructs inside the modules use very simplified pseudocode.

4.1. Image data sources & application that uses PfeLib

We begin the description using Figure 5 in a top-down manner and start with the application [1] that uses PfeLib. In its final field of application this program will acquire images from a hardware device [2], for example, a digital camera over a data path Ⓐ. However, during development and testing it is much more advantageous to use an alternative image source that is able to "emulate" a camera and to deliver standardized test images. PfeRtdxHost [3] can overtake this role of sending and receiving images Ⓑ during development and test phases independently from external hardware. PfeRtdxHost is described in Section 3. Our CV application has now acquired an input image and it moves the program flow to a function that actually does the image processing Ⓒ. This function (we assume its name is `CvAlgo()`) calls various PfeLib routines, such as basic support routines Ⓔ from the basic PfeLib library package [5], or some of the platform-specific performance-optimized image processing routines in PfeImg_xx.lib [7] via Ⓓ.

4.2. Existing library components

We will now discuss the core components of PfeLib in the middle of Figure 5. The top level include file [4] provides (together with some of ancillary .h files) access to the basic data structures, definitions, and function prototypes of PfeLib. The blocks [5], [6], and [7] are the main library modules. The "xx" in the module name stands as a placeholder for an identifier of the target platform for which the library is compiled. As stated above the PfeBase_xx.lib module [5] contains basic support routines and the functions for communication with PfeRtdxHost. The performance-optimized image processing and computer vision algorithms reside in module [7], named PfeImg_xx.lib. Almost all PfeLib algorithms can be driven in the ROS-DMA double buffering mode that will be described in Section 6. In this case, routines of module [6] will be invoked almost completely transparently to the application program.

Every optimized image processing algorithm inside the library module [7] is organized in a systematic manner. At the current, quite early stage of the project (April 2006), approximately 30 image processing algorithms are realized this way, but this number is rapidly increasing. The lower line of blocks in Figure 5 shows the components that make up one PfeLib algorithm. It is very important to give users of PfeLib the opportunity to extend the library with additional functions. So the methodology presented in Section 4.3 can be applied in the same way for any of the following cases.

(a) The designer of an application that uses PfeLib and who needs an additional function that is not yet inside of PfeLib.
(b) A programmer who intentionally wants to add a new algorithm into PfeLib.
(c) Anyone who wants to realize a performance-optimized version of an algorithm for a certain hardware platform that currently exists only in a nonoptimized, generic version.

As an example, we will now create a new algorithm called `PfeFoo()` that will be implemented using our library framework. All components that belong to `PfeFoo()` are depicted in the lower third of Figure 5.

4.3. Components of a new PfeLib algorithm under development

Module [8] with the suffix "`_t`" is a test program that is exclusively dedicated to this algorithm. It manages reading of input and writing of result images from/to PfeRtdxHost. It also includes a test driver that calls the algorithm via Ⓗ. The number of CPU cycles that are needed for processing `PfeFoo()` is tracked by a timing utility. Within the test program it is also possible to perform the test runs under various memory configurations to gain insight about the effectiveness of, for example, L2 cache or the ROS-DMA method.

Module [9], `PfeFoo.c`, contains two functions. Firstly a so-called *API-function* and secondly a generic implementation of the so-called *kernel function* which is also called the *functional behavior*. The API function is the one that is actually called from within an application and therefore it is the only function that has to be visible from the application programmer's point of view. It does not contain the implementation of the algorithm itself. In fact it provides a unique application program interface (API) among several platform-specific implementations of the algorithm. At first it makes some validity checks of the passed function arguments. Then

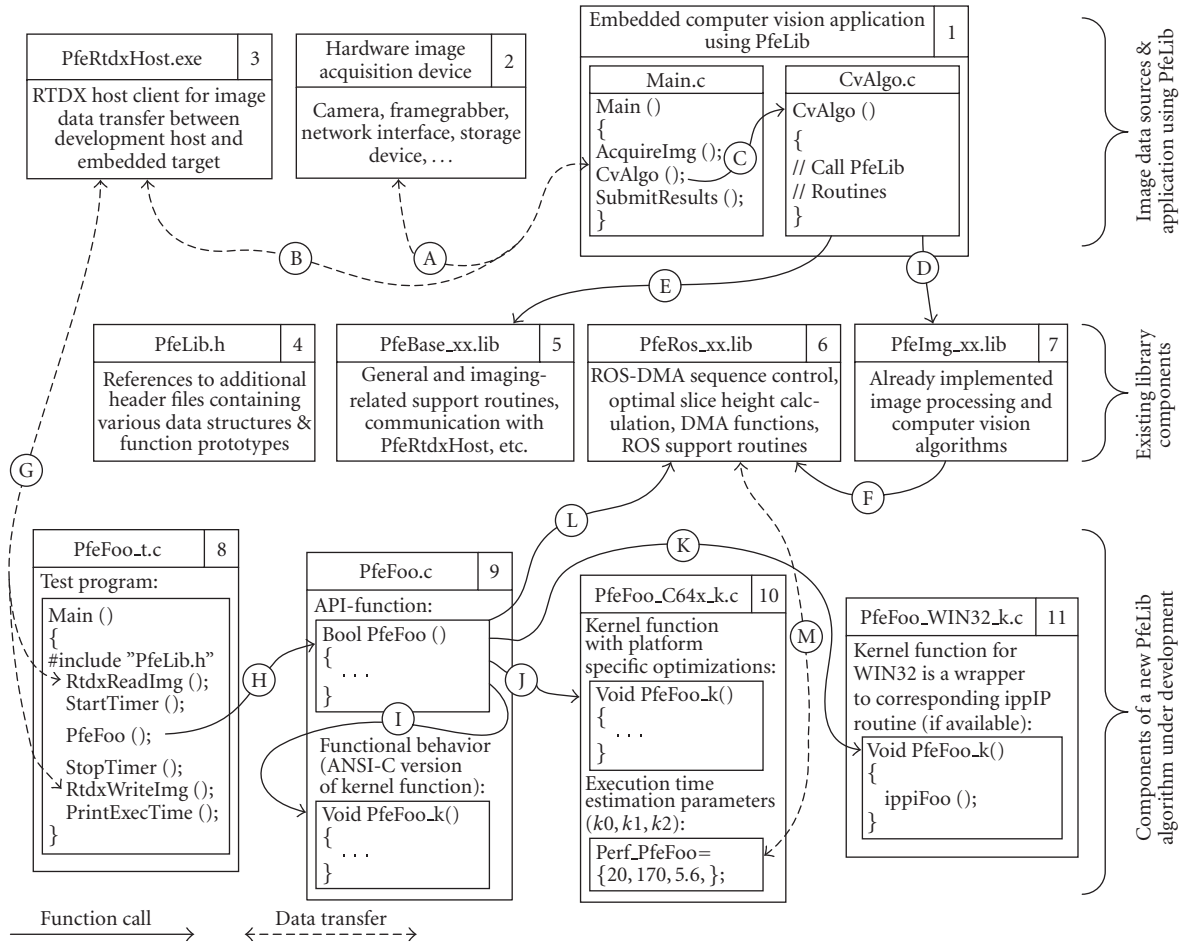


Figure 5: Framework of PfeLib.

it dispatches the program flow to the proper implementation (this happens already at compile time) and furthermore it decides whether to call the kernel function once directly or repeatedly in a loop within a ROS-DMA sequence (this decision is done at runtime). In the latter case, ROS-DMA support routines are used (Ⓛ). Generally, an API-function consists only of few lines of code.

The second function in `PfeFoo.c` is the functional behavior. As it is an implementation of the kernel function for this algorithm, it is suffixed with "`_k`." It is a nonoptimized ANSI-C implementation of the algorithm that can be compiled at any platform. The functional behavior is not intended to run fast but it will provide a reference implementation of the algorithm in a correct and easy to read way. If an optimized and platform-specific kernel function like [10] and [11] is not yet available, or if it is explicitly intended to use the functional behavior, it is called from the API-function via Ⓘ. At this stage the algorithm `PfeFoo()` can already be tested for functional correctness and it can be used by applications—in this case the runtime performance is not important.

Finally, the remaining modules [10] and [11] are performance-optimized versions of the kernel function for specific platforms. Our example in Figure 5 shows one for TI's TMS320C64x DSP platform and one for WIN32. They are called from the API-function via Ⓙ or Ⓚ. The DSP version [10] contains code that resulted from a similar optimization procedure as described in Section 5. We will see there that the execution speed of the optimized version can be much faster, when compared to the functional behavior code. But usually such code uses constructs such as compiler intrinsics or inline assembly that make it incompatible to other platforms. An implementation completely in assembler is also possible. A last item inside of `PfeFoo_C64x_k.c` has to be explained, namely, the execution time estimation parameters k0, k1, and k2. These parameters enable execution time estimates of the kernel function and thus are vital information for the ROS-DMA logic. Their exact meaning is described in [1].

Block [11] in Figure 5 is an example for a kernel function on the WIN32 platform when a corresponding function is already available in the ippIP. In this case the kernel function simply acts as wrapper to that ippIP function.

It has proved to be very practical to prepare a set of template files for the modules [8], [9], [10], and [11], that contain a code skeleton with dummy function names. This set of

template files can additionally contain project- or makefiles for the test programs. Creating a new PfeLib algorithm can then be done by copying and renaming the files and replacing the dummy function name with the desired one. This all can be automated, for example, by scripts. The remaining task is writing the functional behavior and making some additional modifications if necessary. Therefore, we conclude that adding new functionality to PfeLib can be done rather quickly. Creating a highly performance optimized kernel function is more a matter of expertise about a certain hardware platform, and that may take some more effort.

5. PERFORMANCE OPTIMIZATIONS FOR THE C6000 DSP PLATFORM

Although TI's C compiler has sophisticated optimization capabilities [13], programs generated from ordinary C code often suffer from poor performance. The goal of performance optimizations in C is therefore either to reformulate algorithms in such a way that they have less computational strength or to remove constructs that are hindering the compiler in performing better optimizations. We want to find clues about how much difference in execution speed may exist between an ordinary implementation and another one that takes into consideration some specifics of the compiler and the hardware architecture. Therefore, we will describe typical optimization procedures for two image processing functions. A nonoptimized, generic implementation of an algorithm in ANSI-C, called *functional behavior*, is used as a starting point for each function. Then, optimizations are applied in several iterations and their impact is analyzed by measuring the execution times.

Measurement of execution time is done by using one of the hardware timers of the processors. These timers are also modeled by TI's *Device Cycle Accurate Simulators* and therefore we are able to obtain reliable cycle counts in the same way on the simulator as well as on the hardware itself.

During the optimizations described in this section all programs were run in a memory configuration with all the code and data in internal RAM, which is the optimal setting for highest performance.

5.1. Bayer filter demosaicing

The first algorithm to be optimized is a Bayer filter demosaicing function with linear interpolation. Bayer filters are commonly used in single sensor color cameras to add color information to the raw pixel data produced by the sensor [19]. Figure 6 shows a typical filter pattern where every pixel is covered with a filter that is permeable for light with one of the base colors red, green, or blue. To get the RGB values for a destination pixel out of the raw image, the color ID of the current pixel has to be identified. The assignment of the color ID to a pixel is done according to Figure 6.

Equations (1)–(4) (according to [20]) define the behavior of the demosaicing algorithm. Missing color information is

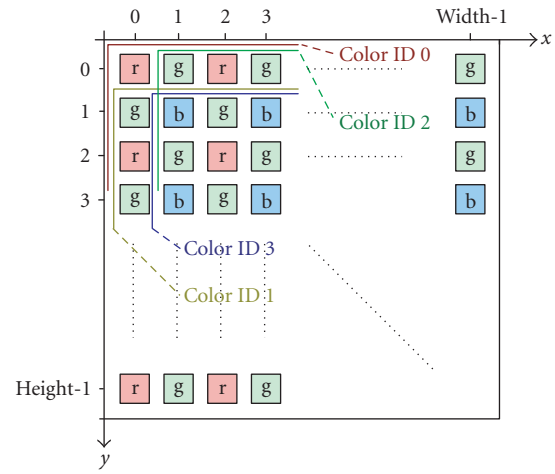


FIGURE 6: Typical Bayer mosaic filter pattern. Pixels marked with r are sensitive for red, g green, and b blue light only.

interpolated from neighboring pixels depending on the colorID of the current pixel

$$\text{colID} = 0 \begin{cases} R_{x,y} = p_{x,y}, \\ G_{x,y} = \dfrac{p_{x-1,y} + p_{x+1,y} + p_{x,y-1} + p_{x,y+1}}{4}, \\ B_{x,y} = \dfrac{p_{x-1,y-1} + p_{x+1,y-1} + p_{x-1,y+1} + p_{x+1,y-1}}{4}, \end{cases} \tag{1}$$

$$\text{colID} = 1 \begin{cases} R_{x,y} = \dfrac{p_{x,y-1} + p_{x,y+1}}{2}, \\ G_{x,y} = p_{x,y}, \\ B_{x,y} = \dfrac{p_{x-1,y} + p_{x+1,y}}{2}, \end{cases} \tag{2}$$

$$\text{colID} = 2 \begin{cases} R_{x,y} = \dfrac{p_{x-1,y} + p_{x+1,y}}{2}, \\ G_{x,y} = p_{x,y}, \\ B_{x,y} = \dfrac{p_{x,y-1} + p_{x,y+1}}{2}, \end{cases} \tag{3}$$

$$\text{colID} = 3 \begin{cases} R_{x,y} = \dfrac{p_{x-1,y-1} + p_{x+1,y-1} + p_{x-1,y+1} + p_{x+1,y-1}}{4}, \\ G_{x,y} = \dfrac{p_{x-1,y} + p_{x+1,y} + p_{x,y-1} + p_{x,y+1}}{4}, \\ B_{x,y} = p_{x,y}. \end{cases} \tag{4}$$

At first, a straight forward implementation of that algorithm is written in C. Algorithm 1 outlines this using of a very simplified, C-style pseudocode. It is important to mention that the support functions `PfeGetPixIndex()` for reading a raw pixel value from the source image and `PfeSetPixRGB()` for assigning a color to a destination pixel are used by this implementation.

Test runs yielded a performance value of 257 cycles per pixel. Although the compiler options were set to `-o2` (optimization at function level), and optimization setting to

```
function PfeBayerLinear()
{
  for y = 0 to image height // loop over rows
  {
    for x = 0 to image width // loop over pixels of row
    {
      colID = colorID according to current x and y
      switch (colID)
      {
        case 0:
          RGB = evaluation of (1) using
                PfeGetPixIndex()
        case 1:
          RGB = evaluation of (2) using
                PfeGetPixIndex()
        case 2:
          RGB = evaluation of (3) using
                PfeGetPixIndex()
        case 3:
          RGB = evaluation of (4) using
                PfeGetPixIndex()
      }
      // set RGB value to the destination pixel
      PfeSetPixRGB(x, y, RGB);
    }
  }
}
```

Algorithm 1: Pseudocode of first implementation.

trade speed for code size was turned on (`-ms`), the result was very disappointing. But it is a starting point for repeated analyzing of the reasons for the poor execution speed and the resulting improvements.

The following optimization steps for this function have been applied for the TMS320C6713 DSP.

(1) Subfunction inlining

Analyzing the number of occurring calls to subfunctions gives an average count of seven calls of `PfeGetPixIndex()` and one call of `PfeSetPixRGB()` per pixel. Calling subfunctions introduces a number of overheads that can be avoided if function inlining is forced. The optimizing compiler is able to perform an automatic function inlining, but there exist several conditions that potentially disable function inlining. During this optimization step the subfunctions were explicitly declared as `inline` and their implementation was moved to a header file to enable function inlining across different source modules. As expected, this measure significantly reduced the execution time. Achieved speedup: 3.82.

(2) Direct access to pixel buffers

During this step the subfunction calls are eliminated altogether, especially because the repeated references to the now inlined `PfeGetPixIndex()` cause a number of address calculations to be performed redundantly. Now, for every iteration of the outer loop over the pixel rows, pointers to the current pixel row and the adjacent rows above and below that are calculated and repeatedly used while iterating over the pixels of one row. Accessing pixel values can now be done with a single de-referencing operation similar to `pixelval=pLine[x]`. Achieved speedup: 1.84.

(3) Elimination of branches in inner loop

One of the most powerful optimization measures that a compiler can apply on VLIW architectures is *software pipelining* of the inner loop. Unfortunately, there exist many reasons why the compiler is prevented from converting a sequential loop code into a software pipelined one. In the current example, the hindering element is the switch statement that produces branches in the compiled code for the inner loop. During this optimization step the switch statement was eliminated as follows. Figure 6 shows that there are two kinds of pixel rows: the rows containing red-sensitive pixels (they have even y-coordinates), and the rows with blue-sensitive pixels which have odd y-coordinates. Within each row, if pixels are traversed in an order of increasing x-coordinates, always the same sequence of pixel-pairs occurs: either rg–rg–rg ..., or gb–gb–gb ..., respectively. This pattern can be used to create different inner loops for the even and the odd rows. Each inner loop walks over pairs of pixels which are processed sequentially and therefore no conditional code appears anymore within the inner loop. Achieved speedup: 1.46.

(4) The `restrict` keyword

The demosaicing algorithm must load a number of pixel values into registers in order to calculate the color of a pixel. Many of them could be reused for calculating the next output pixel if their values are kept in their registers. The reason why the compiler does not work as expected is a problem called *pointer aliasing* [13, pages 3–38]. The compiler must assume that potentially more than one pointer variables reference the same memory location. Between two reads of the content of the same memory location it could have been changed due to a write access over another (aliased) pointer. Compilers must always be conservative, that is, they must assume the worst cases to keep functional correctness. In our example pointer aliasing need not to be considered, so the only thing that must be done is to tell the compiler that specific pointers are definitely not aliased. This can be achieved explicitly by declaring pointer variables with the `restrict`-keyword like "`UINT8 * restrict pLine`". Achieved speedup: 1.97.

The reported test runs for this algorithm were made on a C6713 DSP with 225 MHz. The achieved execution time for the linear Bayer filter demosaicing of 6.3 cycles per pixel yields a computation time for one full frame of our

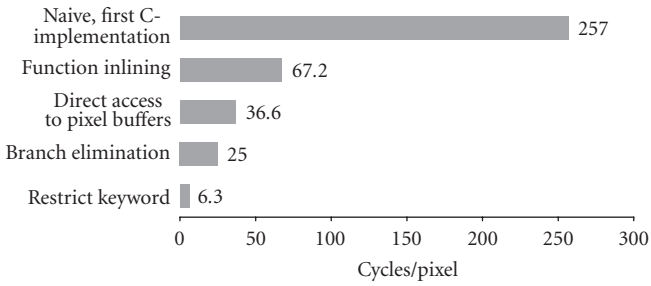


Figure 7: Optimization summary: linear Bayer filter demosaicing.

camera (656×490 pixel) of approximately 9 ms with the C6713@225 MHz. The C6416@1 GHz (the cycle count per pixel is again slightly lower on the C6416 as compared to the C6713) would require about 2 ms. Of course, one could think about making this implementation of the algorithm even faster. It is always a question of the cost-benefit ratio. But in this case it was decided to stop optimization at this stage. Figure 7 illustrates the advances that were achieved in terms of cycles per pixel. Smaller values mean faster execution.

5.2. General linear filter with rectangular mask

We will now show the impact of further optimization strategies on a function that performs a general linear filtering with a rectangular filter mask and a common divisor. A number of commonly used operations can be realized with general linear filtering of images, like image smoothing and sharpening, Sobel or Prewitt edge detection (first derivative operators), Laplacian filters (2nd derivative operator) and many other [21]. The filter mask size of this function and its coefficient values are arbitrary at runtime. Source and destination data are 8-bit grayscale pixel images. The function works according to (5), where $S_{x,y}$ and $D_{x,y}$ are pixel values of the source and destination image, respectively, and x and y are spatial coordinates of a pixel. $M_{i,j}$ are the filter mask coefficients, and w and h denote the width and height of the filter mask

$$D_{x,y} = \frac{1}{\text{divisor}} \sum_{j=0}^{h-1} \sum_{i=0}^{w-1} \left(S_{x-x_a+i,y-y_a+j} \cdot M_{i,j}\right). \quad (5)$$

Again, a generic ANSI-C formulation of the algorithm served as a starting point for performance optimizations. As a result of the former optimization procedure of `PfeBayerLinear()` the Set/GetPix() functions are already forced to be inlined. A test run on the TMS320C6416 DSP with a filter mask size of 3×3 takes 242 cycles per pixel. So we start a similar optimization procedure as before with the following steps.

(1) Direct buffer access and branch elimination

As we learned from the previous example, calls into the subfunctions for accessing to pixel values should be eliminated. This was done here in a similar way. Additionally, the saturating arithmetic was realized more efficiently without a branch. Achieved speedup: 1.35.

(2) Loop coalescing

The innermost loop of this algorithm iterates over the width of the filter mask—usually a rather small figure, in the current example it is three. This is a situation where software pipelining achieves almost no advantage. Furthermore, because software pipelining is only possible for the innermost loop, this powerful instrument is almost useless. A possible solution is loop coalescing. Instead of nested loops that iterate hierarchically over the height and width of the filter mask, only one loop iterates *height* · *width* times. Software pipelining this single loop yields much better efficiency. Achieved speedup: 2.15.

(3) Division by reciprocal multiplication

For each destination pixel, a division by the common divisor must be performed. The processor used has no dedicated HW-unit for integer division. Thus, divisions are a costly operation that take approximately $18 \sim 42$ cycles on C6000 DSPs. As many values have to be divided by the same divisor, a method of division by reciprocal multiplication can be applied. The special task here is to calculate the quotient of two signed 16-bit integers and to return the result saturated to an unsigned 8-bit value. The fact that only the saturated 8-bit result is needed can be exploited for a quite fast implementation on the C64x. We evolved the procedure shown in Algorithm 2 for calculating the reciprocal value. This reciprocal value must be calculated only once in advance. Afterwards, every division with the same divisor including a subsequent saturation to $[0, 255]$ can be done by the sequence shown in Algorithm 3. The C64x specific intrinsic `_mpylir()` performs a signed 16- by 32-bit multiply including a shift-right of the result by 15 bits. `_spacku4()` does the saturation. This method was proved to be correct with exhaustive testing in the range of $[-32738, 32737]$ for both, dividend and divisor in any combination. Achieved speedup: 1.38.

(4) Packed data processing and loop unrolling

TI's C-compiler enables access to very specialized CPU instructions via *compiler intrinsics*. This opens possibilities that are usually only available when programming assembler. The advantages are that the programmer can stay within the high level language and the optimization capabilities of the C compiler are not affected. Among these intrinsics there are many SIMD (single instruction-multiple data) instructions that can process multiple small, for example, 8- bit wide, data values within one cycle when they are packed into one 32-bit register. This packed data processing additionally raises the throughput of the CPU. Secondly, feedback generated by the compiler was used to support its attempts on software pipelining by applying some sort of loop unrolling. Achieved speedup: 4.07.

```
// prepare fast integer division: calc. reciprocal
if (i32Divisor == 1)
  i32Recipr = INT_MAX; // biggest signed int
else if (i32Divisor == −1)
  i32Reciprocal = INT_MIN; // smallest signed int
else if (i32Divisor > 0)
  i32Recipr = ((UINT32)0 × 1 << 31)/i32Divisor + 1;
else
  i32Recipr = −(((UINT32)0 × 1<<31)/-i32Divisor+1);
```

Algorithm 2

```
// =>(dividend ∗ reciprocal) >> 15
tmp = _mpylir(dividend, i32Reciprocal);
// shift right and saturate to 8 u
quotient_sat = _spacku4 (0, tmp >> 16);
```

Algorithm 3

Figure 8 summarizes the optimization steps in terms of consumed processor cycles per image pixel for a 512 pixel wide test image with a 3×3 filter mask. The overall achieved enhancement from the functional behavior to the final version is a factor of 16.33. This final version is still written in C, but the code has become platform-dependent because of the heavy use of compiler intrinsics.

6. DMA BUFFERING

In the field of embedded computer vision, problems will almost certainly arise from the limitations of on-chip memory. Depending on the processor model, in many cases not even one complete image frame fits into IRAM. What is needed within PfeLib is a technique that provides the fastest processing possible, even if image data resides in the much slower external memory.

After an analysis of the problem, a method for DMA (DMA—direct memory access) double buffering in the image processing domain was developed. We call this technique *resource optimized slicing (ROS-DMA)*. It is intended to be a replacement for using level 2 cache in order to gain better performance. An in-depth presentation of this method was published in [1], so we will give only a very brief overview about it here.

The basic idea of ROS-DMA is that images are divided into slices which are transferred from external to internal memory by DMA. The processing in IRAM can be done in parallel to the DMA transfers. Efficient timing is achieved by considering processor-, as well as algorithm-specific parameters. Figure 9 illustrates the underlying concept.

The ROS-DMA method of double buffering within PfeLib has some special properties.

(i) It is a systematic approach for DMA double buffering in the image processing domain—the ROS support routines can be reused for various image processing functions.

(ii) The size of the image slices to be transferred is made variable. A dedicated algorithm computes the optimal height for the first slice (h_f in Figure 9), the "middle" slices h_m, and for the last slice h_l. Therefore, very good performance can be achieved over a wide range of image dimensions.

(iii) The method is superior in runtime performance compared to using L2 cache in almost any case.

(iv) ROS-DMA is better suited for predicting execution times, which is important for real-time applications.

(v) It is easy to use ROS-DMA within an application. It takes not much more than one additional function argument for the call to the PfeLib algorithm.

A case study with the already known performance-optimized PfeLib function `PfeBayerLinearR()` will give an insight about the efficiency of this method. Various test runs have been done on the C6416 device cycle accurate simulator with this function. For each run, a different memory configuration was selected and the number of used processor cycles was counted. Table 2 summarizes the results.

The first configuration, IRAM, is the fastest one, but it is often infeasible in practice because of the small size of on-chip memories of embedded systems. The result of the second configuration, ERAM, is very disappointing. At 860 cycles per pixel the operation has become so slow that this configuration is not applicable in practice. Thus, 64 Kbyte of internal RAM were configured to serve as L2 cache in the third configuration of Table 2. Although the L2CACHE configuration performs much better than the former ERAM config, the achieved performance of 18.8 cycles per pixel is still more than three times slower than the IRAM configuration. In the final test run the ROS-DMA method was activated with an intermediate IRAM buffer of 64 Kbyte. Although image data still resided in ERAM, the operation could be performed almost as fast as in the first run in IRAM configuration.

The ROS-DMA mode of processing within PfeLib can be easily employed for a certain class of image processing operations. The most important thing is that the mapping between source and destination pixels is trivial. If an image processing function meets this requirement, the method is relatively flexible and supports functions with the following source and destination image configurations.

(1S) Operation takes one source image, where only read accesses will occur. No destination image is generated (e.g., the result is a scalar value).

(2S) Two source images, no destination image. An example for this type could be an SSD—(sum of squared differences) function.

(IP) In-place operation. Source and destination images are the same. The only operand image will face both read and write accesses.

(1S1D) Operation with one source and one destination image.

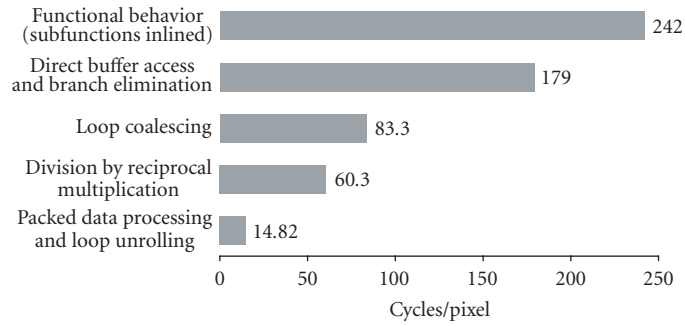


Figure 8: Optimization summary: general linear filtering with common divisor and rectangular filter mask.

Table 2: Memory configuration case study.

Configuration	Description	Performance (cycles/pixel[a])
IRAM	All framebuffers in internal memory. L2-cache is not activated.	6.96
ERAM	Same as IRAM, despite that the framebuffers reside now in external memory.	847
L2 CACHE	Same as ERAM, but 64 Kbyte of internal memory are configured as L2-cache. Cache was reset to a clean state before starting the test run.	19.2
ROS-DMA	Same as ERAM, but 64 Kbyte of internal memory are for an intermediate buffer of the ROS-DMA double-buffering method within PfeLib	7.36

[a]PfeBayerLinearR(), 256×256 pixels, 6416 dev. cyc. acc. simulator, CPU@1 GHz, EMIF@125 MHz.

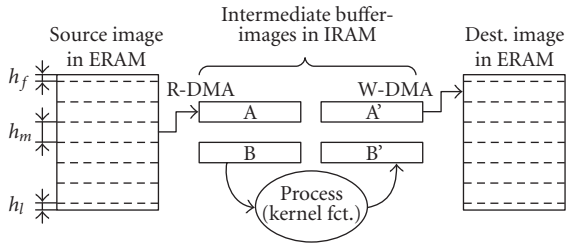


Figure 9: Image slicing with a 1S1D operation.

(2S1D) Operation with two source and one destination image(s).

For the cases listed above the pixel data types (bits per pixel) may vary. Furthermore, operating on regions of interest (ROI) as well as neighborhood operations, like filters, are supported by ROS-DMA.

7. PERFORMANCE TEST RESULTS

This section presents several results of performance tests that have been done with PfeLib. The first part shows diagrams of execution times on a DSP platform over various image dimensions under several memory configurations.

The second part of this section provides a brief cross-platform comparison between PfeLib routines on a C6416 DSP and corresponding routines of the ippIP on an Intel PC platform.

7.1. Different memory configurations on C6416

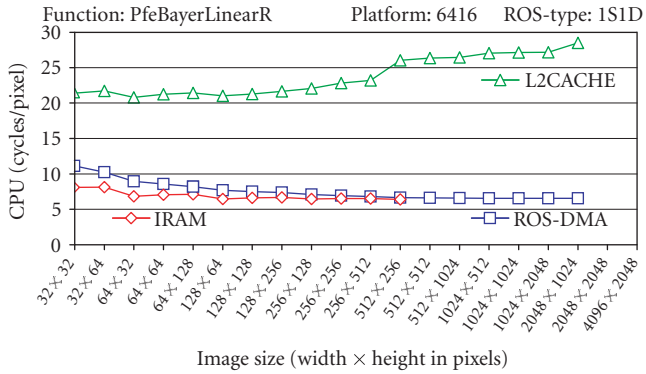
We created a test application for the DSP that allows automatic execution of PfeLib functions on various image dimensions under several memory configurations. PfeLib features switching between these configurations during runtime. The metrics used are cycles per pixel, so a lower value means faster execution. Three memory configurations are plotted.

IRAM with all data in on-chip memory. In this configuration the image dimensions are restricted according to the available on-chip memory.

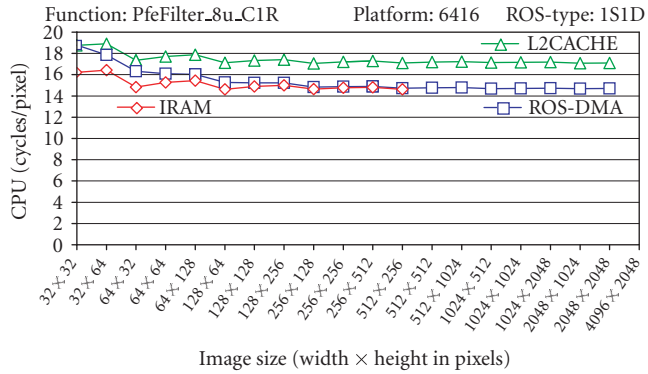
L2CACHE with data in ERAM and 64 Kbyte of on-chip memory configured as L2 cache. L2 cache is initialized to a clean state by performing a *write-back and invalidate* command just before the image processing function was started.

ROS-DMA also with data in ERAM, but with DMA buffering activated instead of L2 cache.

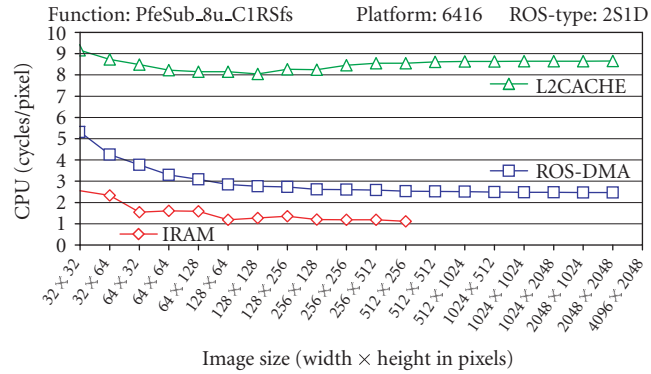
All tests were executed on the C6416 DSK board with 1 GHz running on real hardware. The tests assume a single task environment, no interrupts and no concurrent memory accesses. Figure 10 shows the performance diagrams of some



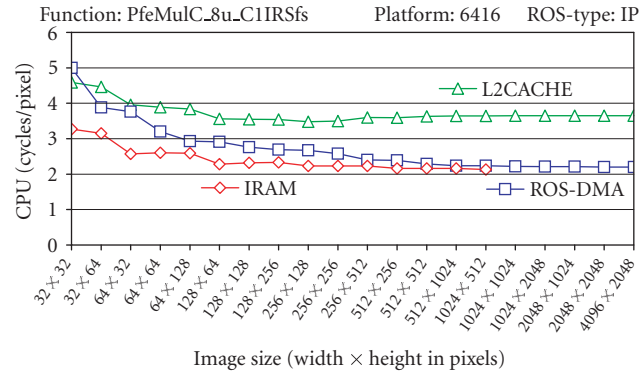
(a) Function optimized in Section 5.1 of type 1S1D with 8 bits per pixel (bpp) input, 24 bpp output and a quite fast kernel function with 6 ∼ 7 cycles per pixel (cpp). ROS-DMA is much faster than L2CACHE and enables processing of images in ERAM with less overhead.



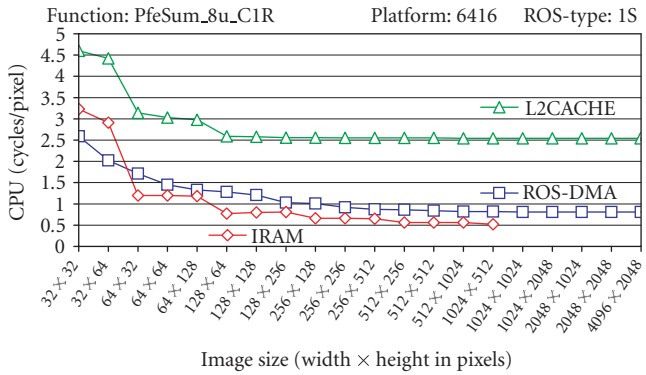
(b) Filter function with a 3 × 3 mask optimized in Section 5.2 with a slower kernel function: ROS-DMA still outperforms L2CACHE, but with a smaller difference.



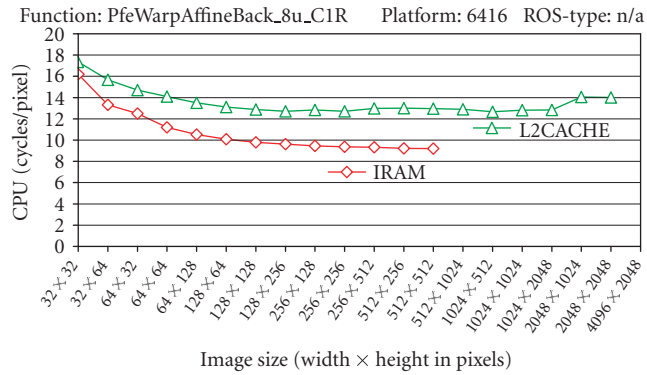
(c) A 2S1D operation where the kernel function is faster than data can be delivered per DMA (memory bound operation); ROS-DMA performance is bounded by the DMA data transfer rate. L2CACHE is significantly slower.



(d) An in-place operation, where the result pixels are written back to the same location where they were read from, is a more favorable configuration for L2CACHE, although ROS-DMA still has an advantage.

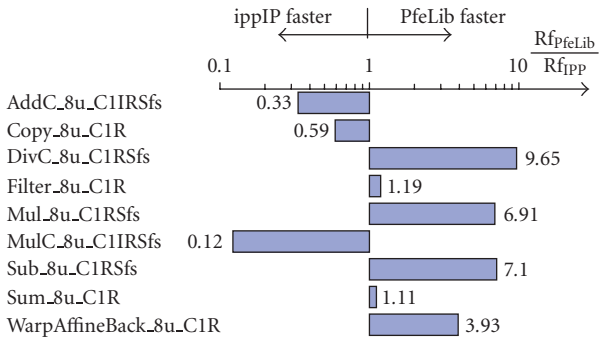


(e) A 1S operation, where ROS-DMA is again significantly faster than L2CACHE. For very small image sizes ROS-DMA is even faster than IRAM because it implicitly converts a 32 × 32 image into a 1024 × 1 image and thus saves on overhead in the kernel function.



(f) This warping operation with linear interpolation has a nontrivial mapping between source and destination pixels. ROS-DMA is currently not available for this type of algorithms. L2CACHE overhead is not very large, but it could be much higher with certain values for the warping coefficients.

Figure 10: Commented performance test results on the C6416 DSK.



PfeLib on C6416 @1GHz; ippIP on Pentium III @1GHz;
image dimensions: 512 × 256

Figure 11: Performance comparison PfeLib—ippIP.

algorithms of PfeLib. The caption texts of the subfigures give some interpretation on the results.

7.2. Performance comparison PfeLib—ippIP

Finally, we want to compare the performance of PfeLib on the DSP platform with the Intel Integrated Performance Primitives (ippIP) library of version 4.1.2 [6], which is only available on the PC platform. Algorithms have been chosen that exist for both libraries. We wanted to compare platforms with similar clock frequencies, therefore we chose an Intel Pentium 3 PC at 1 GHz as a basis of comparison for the C6416 DSK with 1 GHz. The program used on the PC for testing is the demo program shipped with the ippIP called `ippiDemo.exe` (V.4.1).

In some aspect it is problematic to compare a DSP platform with desktop PC processors. Today there exist faster PCs and a state of the art PC with > 3 GHz will probably outperform PfeLib on a C6416 DSP for many situations. But the power dissipation of such a desktop CPU is almost two orders of magnitude higher than that of the DSP used. Additionally, PC processors contain specific instruction set extensions such as SSE which also give them powerful signal processing capabilities. Especially the ippIP routines are often hand-optimized assembly code that make heavy use of SSE instructions.

Our intention was to give an overall comparison in terms of the "cycle efficiency" of different image processing implementations on different hardware platforms. We will consider the diagram in Figure 11 in this context. Figure 11 plots the ratio between the achieved frame rates of various functions. On the DSP, the fastest possible memory configuration was used (mostly IRAM). On the PC, which relies on its data caches, only average performance values are given, so in the worst case an operation on the PC may take significantly longer. It can be seen that the performance ratio varies much among different functions. With very simple functions it can have already a big influence whether a target platform offers one single "good fitting" SIMD instruction for the given task. This may explain the large interval of occurring performance ratios in Figure 11 ranging from 0.12 to 9.65.

8. CONCLUSION AND FUTURE WORK

PfeLib is a software library that incorporates a number of new concepts and developments which in sum make up the novelty of the whole approach in the field of embedded image processing and computer vision. The framework of the library is designed to support multiple (embedded-) platforms as well as it provides the possibility for user-specific extensions. This kind of open architecture gives us hope that PfeLib will find many applications among developers of embedded computer vision solutions.

An early requirement was to supplement the development environment with PfeRtdxHost in order to ease software development on an embedded vision platform. We gave an example where PfeRtdxHost is able to uncover an implementation error with its visual viewing modes that could hardly be found with bitwise comparison against a reference image.

The feasibility of innovative computer vision algorithms often depends on whether they can be realized on small and power-efficient embedded systems. CV is usually a very performance critical field of application and it is of great importance to exploit the potential of the target processor as good as possible. We encountered situations where the chosen DSP platform performed relatively poor when we executed image processing functions coded in generic ANSI C. On the other hand we could achieve very remarkable performance enhancements when we stepped through an iterative optimization procedure that addressed certain specifics of the target platform. Therefore, we believe that it is one of PfeLib's strengths that its framework is designed to provide both, a platform independent API as well as platform specific performance optimizations.

A solution for the problem of slow external memories had to be found, because the successful optimization techniques that were applied to a kernel routine would have become useless in real-world applications when image data must be stored in external memory. Therefore, the novel ROS-DMA method for DMA double buffering was developed and integrated into PfeLib. The modularity of ROS-DMA makes it possible that double buffering can easily be used by newly created image processing functions, because the concept separates code for the memory management from the image processing code. It could be shown that the ROS-DMA method enables faster image processing than using level 2 cache under almost any circumstances.

The challenging work on PfeLib offers many aspects that are worth for further investigations and developments. We want to give here only a selection of the most important items.

(i) Integrating additional algorithms into PfeLib

We think that the currently available set of image processing functions inside of PfeLib enabled a good proof of the overall concept. It will be necessary to add many additional functions to give an application developer a basic set of high-performance functions. On the other hand, the variety of possible image processing operations is so large that no

library can cover everything. Therefore, PfeLib will always be in a state of expansion and it will contain a mixture of algorithms that exist only in a nonoptimized functional behavior version while others are already extensively optimized for certain target platforms.

(ii) Adding execution time estimation functionality

A possible addition to PfeLib would be a function that returns an a priori estimation about the needed CPU cycles for a certain image processing operation. This should be possible with good accuracy when PfeLib is used in IRAM as well as in ROS-DMA memory configuration. Such an extension would be very helpful in realizing dependable embedded real-time vision systems that use dynamic scheduling.

(iii) Extending PfeRtdxHost

Currently, PfeRtdxHost supports only TI DSPs via RTDX. First activities have already been started to extend this tool for other data transfer mechanisms for other target platforms. For instance, TCP sockets could enable image data transfer to embedded PC platforms.

(iv) Extension of ROS-DMA

We showed that the ROS-DMA method is very effective on the TI DSP platform. But image processing operations with a nontrivial mapping between source and destination pixels are currently not supported by ROS-DMA. For example, such operations are geometric transformations like warping functions. For these cases it would be necessary to transfer image tiles instead of slices between external and internal memory. Another interesting task would be to port the existing ROS-DMA components to other embedded platforms with a similar memory architecture.

REFERENCES

[1] C. Zinner and W. Kubinger, "ROS-DMA: a DMA double buffering method for embedded image processing with resource optimized slicing," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '06)*, pp. 361–372, San Jose, Calif, USA, April 2006.

[2] S. Borbély, J. Kogler, and W. Kubinger, "In-the-loop simulation of embedded computer vision applications," in *Proceedings of the 9th International IASTED Conference on Software Engineering and Applications (SEA '05)*, Phoenix, Ariz, USA, November 2005.

[3] R. Behringer, W. Travis, R. Daily, et al., "RASCAL - an autonous ground vehicle for desert driving in the DARPA grand challenge 2005," in *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems (ITSC '05)*, St. Louis, Mo, USA, September 2005.

[4] J. Kogler, H. Hemetsberger, B. Alefs, W. Kubinger, and W. Travis, "Embedded stereo vision system for intelligent autonomous vehicles," in *Proceedings of the IEEE Intelligent Vehicle Symposium (IV '06)*, pp. 64–69, Tokyo, Japan, June 2006.

[5] Texas Instruments Incorporated. TMS320C6000 Technical Brief, February 1999. Literature Number: SPRU197D.

[6] Intel Corporation, "Intel Integrated Performance Primitives for Intel Architecture," 2004. Document Number:A70805-014, Version-014.

[7] B. Kisaicanin, "Examples of low-level computer vision on media processors," in *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '05)*, vol. 3, p. 135, San Diego, Calif, USA, June 2005.

[8] S. Qureshi, *Embedded Image Processing on theTMS320C6000™ DSP*, Springer, New York, NY, USA, 1st edition, 2005.

[9] R. Chassaing, *Digital Signal Processing and Applications with the C6713 and C6416 DSK*, John Wiley & Sons, New York, NY, USA, 2005.

[10] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88)*, pp. 318–328, Atlanta, Ga, USA, June 1988.

[11] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

[12] J. Fürtler, K. J. Mayer, W. Krattenthaler, and I. Bajla, "SPOT-Development tool for software pipeline optimization for VLIW-DSPs used in real-time image processing," *Real-Time Imaging*, vol. 9, no. 6, pp. 387–399, 2003.

[13] Texas Instruments Incorporated. TMS320C6000 Optimizing Compiler User's Guide, May 2004. Literature Number: SPRU187L.

[14] Texas Instruments Incorporated. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, June 2005. Literature Number: SPRU732A.

[15] "Spectrum Digital, Incorporated," 12502 Exchange Drive, Suite 440 Stafford, TX. 77477 USA, June 2005.

[16] Texas Instruments Incorporated. DSP/BIOS, RTDX and Host-Target Communications, February 2003. Literature Number: SPRA895.

[17] D. Pizzolato, "CxImage," April 2005, http://www.codeproject.com/bitmap/cximage.asp.

[18] Texas Instruments Incorporated. XDS560 Emulator Technical Reference, April 2002. Literature Number: SPRU589.

[19] D. H. Brainard, "Bayesian method for reconstructing color images from trichromatic samples," in *Proceedings of the IS&T 47th Annual Meeting*, pp. 375–380, Rochester, NY, USA, May 1994.

[20] U. Furtner, "Color processing with Bayer Mosaic sensors," August 2001, http://www.matrixvision.com/support/articles.

[21] M. Sonka, V. Hlavac, and B. Roger, *Image Processing Analysis and Machine Vision*, PWS, Boston, Mass, USA, 2nd edition, 1999.