An Optimized Vision Library Approach for Embedded Systems

Göksel Dedeoğlu, Branislav Kisačanin, Darnell Moore, Vinay Sharma, Andrew Miller
Texas Instruments, Inc., Dallas, TX
goksel@ti.com

Abstract

There is an ever-growing pressure to accelerate computer vision applications on embedded processors for wide-ranging equipment including mobile phones, network cameras, and automotive safety systems. Towards this goal, we propose a software library approach that eases common computational bottlenecks by optimizing over 60 low- and mid-level vision kernels. Optimized for a digital signal processor that is deployed in many embedded image & video processing systems, the library was designed for typical high-performance and low-power requirements. The algorithms are implemented in fixed-point arithmetic and support block-wise partitioning of video frames so that a direct memory access engine can efficiently move data between on-chip and external memory. We highlight the benefits of this library for a baseline video security application, which segments moving foreground objects from a static background. Benchmarks show a ten-fold acceleration over a bit-exact yet unoptimized C language implementation, creating more computational headroom to embed other vision algorithms.

1. Introduction

Combining high compute performance with low power requirements, embedded processors have been playing a central role in equipments such as mobile phones, network cameras, and automotive safety systems. Among the wide range of signal processing needs, computer vision has proven particularly challenging: the underlying algorithms are not only computationally demanding, but they also are still evolving into more robust solutions.

How can we ease the ever-growing pressure to accelerate such an evolving mix of vision applications? In this paper, we take a software library approach and accelerate the most common computational bottlenecks via processor-specific optimizations. We explain how we designed VLIB, a collection of over 60 low-level and mid-level vision kernels for computer vision applications on embedded processors such as Digital Signal Processors (DSP). Through this library, developers can conveniently access optimized DSP performance for pixel-intensive tasks, while still enjoying the flexibility of a C-programmable platform.

2. Requirements for embedded vision

As vision algorithms transition from research labs to product groups, it is often necessary to transform Matlab/Python scripts running on powerful PCs into C programs on embedded system-on-chips. This transition can be difficult, because a cost-effective solution might require a fierce trade-off with respect to algorithm complexity, memory usage, and I/O bandwidth. DSP implementations of several embedded vision applications have been described in [1-3].

As vision algorithms are applied in embedded systems, optimization of vision algorithms [4] and software [5] is critical to achieve high performance. Among the greatest challenges in achieving high code efficiency in an embedded environment is the reformulation of algorithms to exploit the advantages of the fixed-point data representation and direct memory access. These algorithmic changes offer tremendous run-time performance advantages over using floating-point and cache memory, and are considered a must in embedded vision.

2.1. Fixed-point representation

VLIB is implemented entirely in fixed-point arithmetic, wherein we judiciously assigned fractional bits to intermediate and final results of vision algorithms. Consider, for instance, the recursive moving average M of an 8-bit pixel intensity value I,

$$M_{\mathrm{updated}} = (1-w)M_{\mathrm{previous}} + wI_{\mathrm{new}} \quad (1)$$

where w is the weight and the initial condition is $M = M_0$. When applied to a *time-ordered* sequence of pixel intensities I, the variable M would smoothly adapt to the observed sequence of I values. Using a static camera, this

approach can effectively capture a pixel-wise background model of the scene [6]. Eq. (1) can also be applied to a *spatially-ordered* sequence of pixel intensities, e.g., left-to-right (LR) along an image scanline. This would implement the LR component of a horizontal infinite-impulse response filter, which can efficiently approximate high-order convolution filters [7].

Since the adaptation (or blending) rate w in Eq. (1) is a real number between 0 and 1, it is imperative to capture the fractional adaptation steps for a successful numerical implementation. How many bits should we assign to represent the variable w? In other words, what is the smallest w that we would care to implement? Recognizing that the answer would vary from one application to another, we took into account the specific Instruction Set Architecture of the C64x DSP core and designed two different fixed-point versions of Eq. (1) in VLIB: 16-bit and a 32-bit representation of M, both assuming an 8-bit integer value for I. The former uses 7 fractional bits, whereas the latter uses 23 bits.

VLIB functions that support the 16-bit representation for M make optimal use of the two-way Single-Instruction-Multiple-Data (SIMD) instructions of the architecture such as MPY2 and SUB2. These treat the lower- and upper-halves of the 32-bit wide registers as two separate entities and therefore perform twice as many multiplication and subtraction operations per clock cycle. Applications that require more than 16 bits to represent M can fall back to the 32-bit VLIB functions, which will use the native MPY32 and SUB instructions and run relatively slower.

2.2. Designing for direct memory access

Given the limited amount of on-chip memory, the library supports block-wise partitioning of video frames whenever possible. This is in anticipation of data movements between on-chip and external memory via direct memory access (DMA), which removes data flow responsibilities from the DSP core: while the DSP is busy working on a chunk of the video frame, the DMA engine fetches the next chunk of data to be processed and places it in a buffer. Under this mode of operation, it is often necessary to reserve sections of on-chip memory as *input*, *scratch*, and *output* buffers, and pass their pointers to the function being called. For instance, the prototype of the VLIB function that implements the horizontal IIR filter is

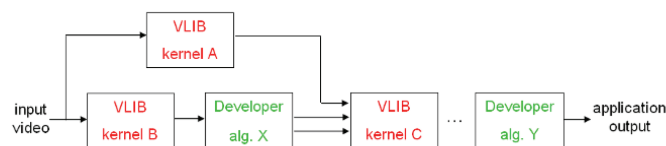


Fig.1. VLIB is a highly granular library: it consists of over 60 kernel functions that are meant to be building blocks of an overall application. This design enables maximum re-use of intermediate results within the application flow to facilitate optimal on-chip memory buffer (re)use. Developers can blend their proprietary algorithms with VLIB or other libraries without incurring any overhead.

```
int VLIB_recursiveFilterHoriz1stOrder(
       char  *out,
       char  *in,
       int   width,
       int   height,
       short weight,
       char  *boundaryLeft,
       char  *boundaryRight,
       char  *buffer);
```

The last three arguments in this function call are most indicative of an embedded implementation: the arrays `boundaryLeft` and `boundaryRight` allow the function to propagate partial results across the scanline, and there is a scratch `buffer` to store intermediate results. With this style of Application Programming Interface (API), VLIB supports composing application flows that automate block-wise data movements and computation [8].

2.3. Considerations on kernel granularity

While vision-based solutions have been enjoying a wider adoption during the past few years, they remain far from being standardized. As vision algorithms keep evolving to deliver better application performance and reliability, it is critical for an optimized library to give the right level of support and flexibility. In this context, we designed VLIB to offer the most common vision functions as building blocks, resulting in a highly granular library of over 60 kernels. As depicted in Fig. 1, this organization enables re-use of intermediate results for optimal on-chip memory buffer usage. In addition, developers can blend their proprietary algorithms with those already in VLIB or other libraries and compose their specific application flow.

3. What are the computational bottlenecks?

Our analysis revealed that pixel-intensive tasks such as foreground-background segmentation, color conversion, and low-level feature computations account for most of the bottlenecks in embedded vision. We therefore designed VLIB to target these computations first, resulting in a library of over 60 low- and mid-level vision kernels.

3.1. Functional overview

In addition to a number of digital signal and image processing functions of the DSPLIB [9] and IMGLIB [10] libraries, VLIB provides functions for the following common vision tasks:

- Color conversion: RGB, YUV (planar, semi-planar, and interleaved), HSL, and LAB.
- Background-foreground segmentation: exponentially- and uniformly-weighted running means and variances, mixture of Gaussians, and the statistical background subtraction methods that use the above models.
- Binary blob analysis: erosion and dilation operations, connected components labeling.
- Gradient and edge analysis: Canny edge detection
- Multi-scale analysis: image and gradient pyramids using box- and Gaussian-filters.
- Recursive Infinite-Impulse Response filters: horizontal and vertical.
- Feature analysis: integral image, Hough transform for lines, weighted histograms for scalars and vectors, Legendre moments, L1 and Bhattacharya distances, Harris corner score, non-maximum suppression.
- Tracking: Lucas-Kanade sparse optical flow, Kalman filter, normal flow, Nelder-Mead simplex search.

3.2. Optimization on the C64x/C64x+ DSP core

When using a highly optimized library approach as in VLIB, it is important to understand the underlying processor architecture in order to achieve the code efficiency goals. We optimized the library for the Texas Instruments C64x/C64x+ DSP cores, which have eight parallel functional units with single-instruction-multiple-data capabilities (Fig. 2). For optimization, we analyzed and fine-tuned most pertinent loops through C-intrinsics and maximized the throughputs of software pipelines whenever possible [11].

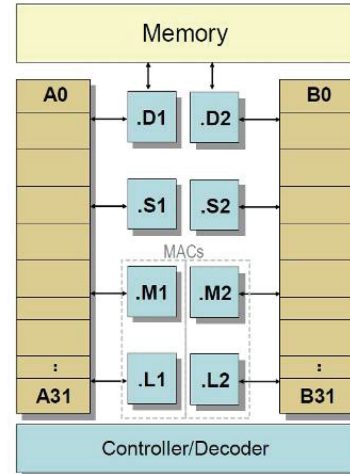
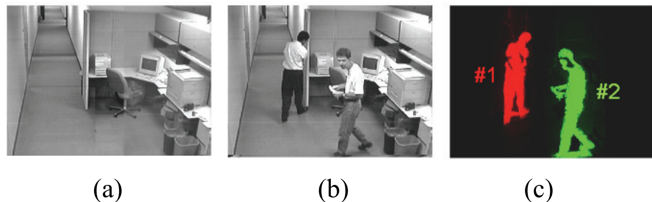


Fig.2. C64x and C64x+ are 8-way VLIW cores that can execute up to 8 instructions in parallel. Furthermore, SIMD extensions enable yet another layer of parallelism, leading to highly-optimized code for pixel-intensive vision tasks.

4. Demonstration in video analytics

We highlight the benefits of the optimized library by constructing a baseline application commonly used in video analytics systems. The application, which we call Moving Object Segmentation (MOS), extracts foreground objects from a stationary background and assigns them unique labels. The application maintains and updates a background model to adapt to gradual changes in the scene, and generates foreground object segments, or blobs, as its primary output. In a video analytics system, these blobs serve as input to other higher-level algorithms for tasks such object detection, tracking, and classification.

The MOS application relies on several algorithms that scan the entire image, “touching” every pixel at least once. While conceptually simple, the pixel-intensive nature of these algorithms make this application one of the more computationally expensive processing blocks in a video analytics system. In this section, we describe the construction of MOS utilizing the optimized library, its implementation on a DSP, and the performance benefits over a natural C language implementation of the same application.



(a) (b) (c)

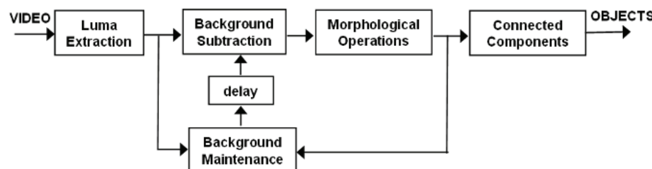


Fig. 3: As a representative video analytics application, Moving Object Segmentation detects moving foreground objects apart from the background scene and assigns them unique labels. (a) The background model has adapted to the empty scene. (b) The input video frame that will be compared against the background model. (c) Labeled foreground masks showing two objects detected.

TABLE 1: WE BENCHMARKED TWO VERSIONS OF THE MOVING OBJECT SEGMENTATION AT 320X240 PIXEL RESOLUTION AT 10 FRAMES/SECOND ON THE DM6437 EVALUATION PLATFORM. COMPARED TO ITS NATURAL-C IMPLEMENTATION, THE VLIB LIBRARY SPEEDS UP THIS APPLICATION BY A FACTOR OF 10.

Moving Object Segmentation VLIB kernel	**C-code MHz**	**VLIB MHz**
VLIB_convertUYUVtoLuma	0.92	0.3
VLIB_subtractBackgroundS16	12.4	1.0
VLIB_erode_bin_square	17.5	0.2
VLIB_dilate_bin_square	17.1	0.2
VLIB_updateEWRMeanS16	13.2	0.9
VLIB_updateEWRVarianceS16	17.8	1.1
VLIB_connectedComponentList	2.9	1.4
Overhead	4.6	4.6
Total	*86.3*	*9.6*

4.1. Moving Object Segmentation

The data flow and processing blocks used in MOS are depicted in Fig. 3. Once the luminance channel of the incoming frame is extracted (Fig. 3b), it is compared against the background model (Fig. 3a) to generate a binary foreground image. Morphological operations such as dilation and erosion are applied to this binary image to eliminate isolated pixels and close holes. Finally, connected components labeling is applied to the cleaned binary image to generate foreground object segments, or blobs (Fig. 3c).

VLIB provides the necessary functionality to construct the MOS application. For background subtraction and maintenance, the library provides a choice of algorithms. For a uni-modal background, the single Gaussian can be approximated either using kernels for exponentially weighted running mean and variance [6], or the simpler uniformly weighted running mean and variance. The kernel for background subtraction provides the statistical differencing using the Mahalanobis distance. For a multi-modal background, the library provides functions for background modeling and subtraction using the Mixture of Gaussians (MoG) approach [12].

While MoG can provide more robust background modeling, it is more expensive than the uni-modal approach. The VLIB MoG kernel operates on the luminance channel and provides a maximum of 3 modes per pixel. For each mode, the kernel maintains the mean, variance and the mixing weight. In addition to being computationally more expensive than the uni-modal approach, the MoG kernel also requires more than 3x the data movement between on-chip and external memory. For the sake of simplicity, we chose to demonstrate the MOS application using the uni-modal background approach.

We list the library functions utilized for each block in Table 1. While the literature on vision-based video surveillance offers more complex background subtraction schemes, these choices reflect the computational tradeoffs in real-world video analytics solutions.

4.2. Implementation on the DM6437 EVM

To measure the performance differences between straightforward, unoptimized kernels written in standard C and those incorporating C64x-specific optimization, we selected the TMS320DM6437, a digital media system-on-chip built around TI's C64x DSP core as our test platform. Developers can often achieve acceptable real-time performance on this powerful DSP by relying on the C6000 DSP Optimizing complier [13] and by simply employing a naïve implementation that relies on the cache to negotiate the data flow between external memory and the processor. To approach the device's full performance potential, however, manual scheduling of data movement using DMA controllers is generally necessary. Moreover, DMA memory management eliminates performance variation caused by cache incoherency, ensuring that results that we measure are consistent.

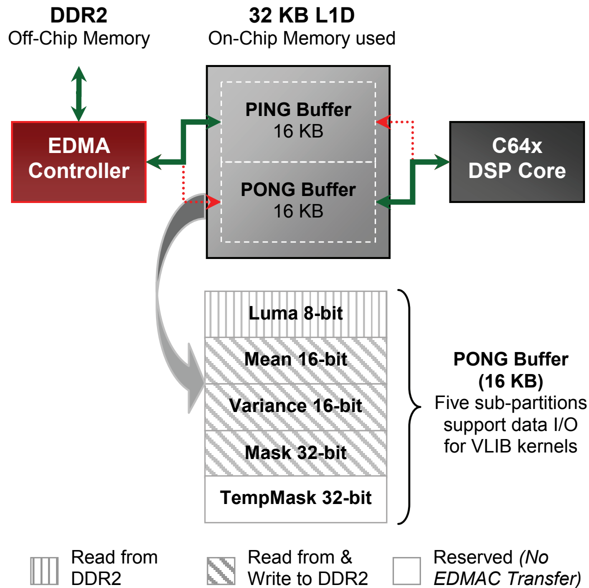


Fig. 4: Fast on-chip L1D memory is partitioned into two 16 KB buffers, "ping" and "pong," each of which is further partitioned to support data I/O for VLIB kernels used to implement MOS. As the EDMAC retrieves data from DDR2 and fills respective sub-partitions with the "ping" buffer, the DSP reads, processes, and writes data to the "pong" buffer. This double-buffering scheme allows DSP to maximize throughput.

We developed a data trafficking approach that uses the DM6437's Enhanced DMA Controller (EDMAC) to move data between external DDR2 and on-chip L1D memory. We first examined the data input and output requirements of VLIB kernels used to implement MOS. We identified consecutive kernels that use the same data, which gave us the opportunity to move data supporting several kernels instead of just one. Each memory block transfer requires DMA overhead and channel bandwidth, both of which can be reduced by careful planning.

As shown in Fig. 4, our scheme uses 32 KB of fast on-chip L1D memory, partitioned into two 16 KB buffers, "ping" and "pong." Each 16 KB buffer is further partitioned to support I/O for five kernels:

- `VLIB_subtractBackgroundS16`,
- `VLIB_erode_bin_square`,
- `VLIB_dilate_bin_square`,
- `VLIB_updateEWRMeanS16`,
- `VLIB_updateEWRVarianceS16`.

As the EDMAC retrieves data from DDR2 and fills respective sub-partitions within the "ping" buffer, the DSP reads, processes, and writes data to the "pong" buffer. When the DSP is done, it signals to the waiting EDMAC, then switches to the "ping" buffer and begins to execute the same five kernels. When the EDMAC receives the signal, it transfers some of the "pong" sub-partitions back to their original locations in DDR2 before replacing all sub-partitions with new data from other locations in DDR2. This double buffering approach prevents the processor from stalling while waiting on data, allowing the DSP to sustain maximum throughput.

4.3. Performance benchmarks

We contrast the performance of a straightforward C language implementation of MOS kernels against our optimized library in Table 1. These ANSI C functions contain no processor-specific intrinsic functions nor do they deploy coding "tricks" used by experts with deep experience with the C compiler. All source code for our comparison was compiled with Texas Instruments' TMS320C6000 Optimizing Compiler with the most aggressive file-level optimization enabled through the -o3 flag. To mimic typical processor loading, we programmed the DM6437 Digital Video Development Platform to process 320x240 pixel (QVGA) videos at the rate of 10 frames per second.

At the application level, the library enables a ten-fold speedup. At the level of individual kernels, as detailed in Table 1, the gains are tightly coupled to the nature of the computation: while pixel-intensive kernels with regular & deterministic access patterns benefit extensively from the library, data-dependent kernels such as Connected Components Labeling enjoy relatively modest speedups. In our implementation, we isolated un-optimized software components that are not required to deliver MOS functionality, e.g. overlays for displaying graphics and text, etc., and labeled them as "overhead."

To ensure repeatability of our measurements, a short uncompressed video sequence was loaded into external DDR2 memory on a DM6437 evaluation module (EVM) in lieu of live video. While the 320x240 (QVGA) sequence was processed at 10 fps, we measured the processing time for each kernel. Using the DM6437's hardware on-screen display engine to render a semitransparent overlay of kernel performance, we programmed switches on the EVM to select between the unoptimized version of the library, e.g. VLIB OFF, and the optimized version, VLIB ON. Figures 5 and 6 are screen shots of the actual MOS implementation with VLIB turned off and on, respectively.

Fig. 5: Screen shot of MOS implementation running on the DM6437 with unoptimized library. The loading on the DSP is approximately 86 MHz for processing a 320x240 pixel video sequence at 10 frames per second.

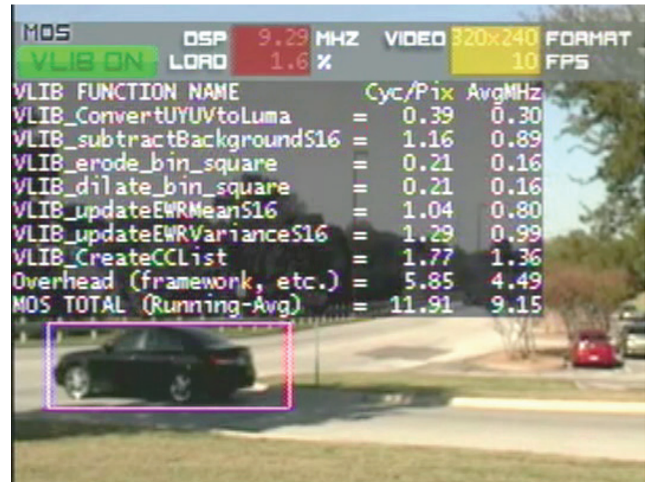


Fig. 6: Screen shot of MOS implementation running on the DM6437 with optimized library. At the application level, VLIB provides about ten-fold speedup in performance.

5. Conclusion

We have explained key aspects of the software library approach used to design VLIB, a new software library that accelerates computer vision applications for high-performance embedded systems. By significantly speeding up pixel-intensive operations, the library provides more headroom for innovative algorithms, and enables processing of more channels at higher resolutions. The library approach is applicable in video analytics, automotive safety and machine vision domains as well as other wide-ranging vision applications.

References

[1] B. Kisacanin, "Examples of Low-Level Computer Vision on Media Processors," Proc. IEEE Workshop on Embedded Computer Vision, 2005.

[2] C. Arth, C. Leistner, and H. Bischof, "An Embedded Platform for Remote Traffic Surveillance," Proc. IEEE Workshop on Embedded Computer Vision, 2006.

[3] C. Arth, F. Limberger, H. Bischof, "Real-Time License Plate Recognition on an Embedded DSP-Platform," Proc. IEEE Workshop on Embedded Computer Vision, 2007.

[4] B. Kisacanin and Z. Nikolic, "Algorithmic and software techniques for embedded vision on programmable processors," Signal Processing: Image Communication, Vol.25, 2010, pp. 352–362.

[5] B. Kisacanin, G. Dedeoglu, D. Moore, V. Sharma, and Z. Nikolic, "DSP Software Libraries for Automotive Vision Applications," Proc. 2009 DSP for In-Vehicle Systems and Safety Workshop, 2009.

[6] K. P. Karmann, and A. von Brandt, "Moving object recognition using an adaptive background memory," in Time-Varying Image Processing and Moving Object Recognition, Elsevier Science, 1990, pp. 289-307.

[7] R. Deriche, "Fast Algorithms for Low-Level Vision", IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 12, No. 1, 1990.

[8] Texas Instruments, "Vision Library (VLIB) Application Programming Interface," TI SPRUG00C, 2009.

[9] Texas Instruments, "TMS320C64x+ DSP Library Programmer's Reference," TI SPRUEB8B, 2006.

[10] Texas Instruments, "TMS320C64x+ DSP Image/Video Processing Library," TI SPRUF30A, 2008.

[11] E. Granston, "Hand-Tuning Loops and Control Code on the TMS320C6000," TI SPRA666, 2006.

[12] C. Stauffer and W. Grimson, "Adaptive background mixture models for real-time tracking", in Proc. of IEEE Computer Vision and Pattern Recognition, 1999.

[13] Texas Instruments, "TMS320C6000 Optimizing Compiler User's Guide," TI SPRU187O, 2008.