Scalable Frame to Block based Automatic Converter for efficient Embedded Vision Processing

Senthil Kumar Yogamani
Texas Instruments
Dallas, TX, USA
ysenthilece@gmail.com

B H Pawan Prasad
Samsung Research
Bangalore, India
mail4pawan2003loud@gmail.com

Rajesh Narasimha
Metaio
Dallas, TX, USA
rajesh.narasimha@gmail.com

Abstract

A typical digital signal processor (DSP) uses hierarchical memory to handle the trade-off between cost and speed. It has a fast on-chip memory with data-access rates similar to the DSP's processing rate but it is not large enough to hold the entire Image data. Image buffers typically reside in the larger external memory like DDR whose data access rate is ~4-6X slower than the processor rate. Cache or direct memory access (DMA) mechanisms are used to improve the slow access rate of external memory using the internal memory. Optimizing an embedded processing application to be efficient for such hierarchical memory systems requires block-based algorithm design. This is usually accomplished by manually re-designing the code. This effort requires several man months and DSP expertise. In this paper, we automate this process and demonstrate a performance improvement of ~2-4X over conventional frame level processing. We believe that the proposed solution is novel in the sense that it is fully automated and scalable to any memory size and speed. We use a compiler assisted parser to extract the relevant function parameters and use them to re-target the code to be block-based and handle memory management automatically. This is an offline code generation process with self-verification. We have implemented and tested the parser for Texas Instruments (TI) C6000 DSPs but the method is generic to work with any processor core.

1. Introduction

DSP uses hierarchical memory to handle the trade-off between cost and speed. Image data typically resides in the cheap and large external memory like DDR whose data access rate is ~6X slower than what the processor can process. The DSP contains on-chip memory such as L1 which is small, expensive and with data rate similar to the DSP's data rate. For example, the clock-rate of C6000 TI DSP processor is 600 *Mhz* and the external memory access rate is 100 *Mhz* as shown in Figure 1. Optimizing an image/vision processing application to be efficient for such hierarchical systems requires re-design of the algorithm flow which requires both systems and algorithms expertise and takes several man months for engineers and requires support from the Silicon vendor. The proposed code-parser which generates functionally equivalent block-based code from frame-level code. The code parser should ideally be included in the compiler or it can be also used as a stand-alone code generator tool.

Conventionally the frame to block level design is addressed in two ways namely manual re-implementation of the code and using a framework to handle block allocation. Manual reimplementation of code requires several man-months of effort and requires special skills [6]. Framework requires effort to wrap the code to the framework format and it also introduces performance overheads. A block based algorithm can be optimized for superior performance and low power when compared to a frame based algorithm. This is mainly because of the fact that block based algorithms can be designed to process smaller blocks of images that can utilize the internal memory of the CPU. Further, this design will also reduce the amount of power because of the optimal usage of available resources on the SoC. There are two notions of optimality: performance and power consumed. For performance, it is sufficient to hide the memory traffic behind processing times. For power consumption, repetition of bringing in the same data has to be avoided. Typically the latter covers performance too.

In this work, we propose a code parser based automatic solution to generate block-based versions of the algorithm. We automate this process and demonstrate a performance improvement of *~2-4X* over conventional frame level processing. We use a compiler assisted parser to extract the relevant function parameters and use them to re-design the code to be block-based and handle memory management automatically. This is an offline code generation process with self-verification. We believe that the proposed solution is novel in the sense that it is fully automated and scalable to any memory size and speed. The proposed method was implemented and tested on the Texas Instruments (TI) C6000 DSPs. Our approach is scalable to different processor cores such as GPU and does

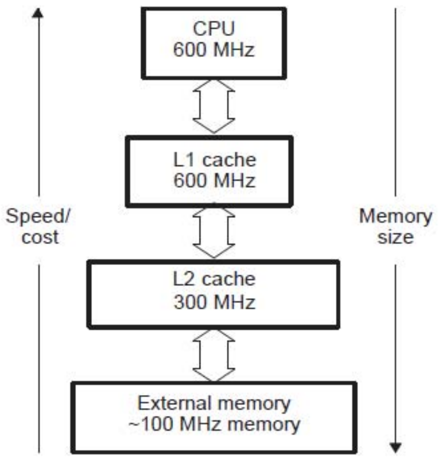


Figure 1: Illustration of Hierarchical Memory in a TI C6000 DSP Processor [2]

not require source code modification of the algorithm nor does it introduce overheads.

Ko *et al* model block-based DSP systems in [1] and discuss automated buffer management in [8] in the context of architecture/tools exploration and not in context of code generation. The reference guide [3] is a documentation of a commercial product which addresses a similar problem. It uses manual parameterization and wrapping of APIs. The details of the implementation are not disclosed in the reference guide [3]. In comparison the proposed method attempts to automate parameterization and does not require wrapping of APIs.

Section 2 introduces the background of the problem and discusses relevant concepts. Section 3 provides the problem description, proposed solution and performance improvements. In Section 4, we discuss a specific use case namely Canny Edge Detection and provide practical sizes of buffer involved in real application.

2. Background

In this section, we provide a description of hierarchical memory systems and discuss the concept of ping-pong buffering.

2.1. Hierarchical memory in Embedded Systems

A "memory hierarchy" in computer storage distinguishes each level in the "hierarchy" by speed, size and cost of memory. For example, in Figure 1, the clock-rate of C6000 TI DSP processor is 600 *Mhz*. The internal memory (on-chip memory) access rates are 600 *Mhz* (L1) and 300 *Mhz* (L2) respectively whereas the external memory access rate is 100 *Mhz*. Internal memories are faster and smaller in physical size but expensive whereas the external memory is cheaper but larger in physical size and lower in speed.

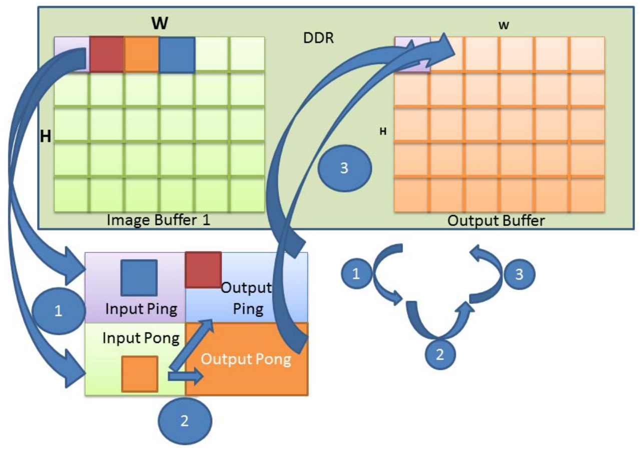


Figure 2: Ping-Pong Buffering. 1. External memory to internal memory data transfer, 2.DSP processing data from input and writing to the output and 3. Internal memory to external memory data transfer.

In this paper, we focus on systems having internal memories that are addressable (not just a cache) which is not possible on ARM cores and external memories. Most of the current generation embedded DSPs fall under this category. Fig 1 shows the memory hierarchy in TI DSP [2].

2.2 Ping-Pong Buffering

Ping-Pong buffering is a programming technique that uses two buffers to speed up a computer that can overlap I/O with processing. Data in one buffer is being processed while the next set of data is read into the other buffer using a separate engine called the DMA. In streaming media applications, the data in one buffer is being sent to the sound card and/or display adapter, while the other buffer is being filled with more data from the source (Internet, local server, etc.). When video is displayed on screen, the data in one buffer is being filled while the data in the other is being displayed. Full-motion video is speeded up when the function of moving the data between buffers is implemented in a hardware circuit rather than being performed by software. In one instance, the DSP is controlling the input ping buffer and after processing the data writes into the output ping buffer. In parallel, the data from external memory is read into the input pong and simultaneously data is written into external memory from the output pong using the DMA controller. In the next instance the ping and pong buffer exchange their roles. This process is depicted in Figure 2.

Input and output data traffic can be skewed due to the difference in buffer sizes and dedicated DMA channels are used for transferring data as shown in Figure 2. The data involving DMA channels leads to significant power consumption relative to internal memory accesses. Power consumption can be minimized by bringing in sufficient data that can be processed and executing as many kernels processing as possible. This process of making use of multiple kernels at the block level is called chaining which is discussed in the later section.

3. Proposed Solution

In this section, we describe the problem and explain the parameters involved for automatic conversion. The required parameters are extracted from the data-flow graph generated by the compiler. From the parameters, we illustrate how the block-based version of the code is generated and performance benchmarks are shown to illustrate the improvements.

3.1. Problem Description

Input/output to the Tool: Input to the tool is an Image processing application consisting of a set of kernels operating on an Image frame. Output is an equivalent Block-based version generated from the input which is optimized in terms of performance and power dissipation. The generated code contains prolog and steady-state code. Steady-state code assumes previous set of inputs are available. For the first iteration, the input buffers are empty and there is no re-use. This step is called prolog.

System Parameters: These parameters are hardware (HW) dependent factors which can be encoded in the tool or it might be already encoded in a HW specific compiler. They correspond to processor clock rate, internal memory size, external memory size and DMA data transfer rate.

Kernel Parameters: These parameters are specific to each Image processing function. Kernel block size refers to the size of the window size required for generating one output pixel. It is the most important parameter based on which the internal buffer sizes are determined. The other buffer size parameters are input buffer size which captures the number and size of the input buffers and output buffer size which captures the number and size of the output buffers. The last attribute is a flag to denote whether the function is block-based or not.

3.1. Compiler assisted Code Parser for extraction of Kernel Parameters

In this section, we describe the methodology to extract the kernel parameters for each kernel present in the image processing application. An optimizing compiler analyzes each kernel and builds a data flow graph (DFG). A simple data flow graph for a dot product kernel is shown in Figure 4. It consists of two inputs that are loaded into internal memory, followed by multiplication and accumulation

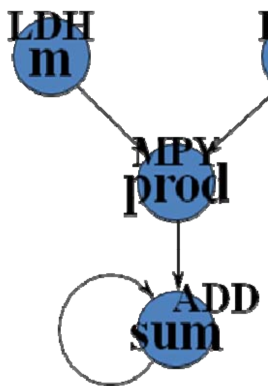


Figure 3: Data-Flow Graph (DFG) generated by TI C6000 compiler

operations. A typical DSP compiler generates software pipeline information using which the DFG data structure is generated. Graphical representation of this DFG is shown in Figure 3.

From the DFG, the appropriate input and output pointers and their sizes can be inferred in a straight forward manner. The source nodes form the inputs and the sink nodes form the output. To find the kernel size we make following assumptions. Kernel size is assumed to be small and their corresponding loops are collapsed so that two nested loops are obtained for traversing rows and columns or a merged version with one single loop. Next, all the input pointer offsets corresponding to one output pixel is collected. The vertical and horizontal lengths of the offsets provide the kernel size parameter. For example, the offsets of a 3x3 Sobel filter are 0, 1, 2, *w, w+2, 2*w, 2*w+1, 2*w+2* (where *w* is the width of the image). The vertical and horizontal lengths are 3 and 3 respectively which correspond to the kernel parameter.

Finally the characteristic of a particular kernel being block based or non block based is determined by the pointer increment used by the DFG. If the pointer increments are bounded constants, then the kernel is considered to be block based, otherwise it is considered as non block based. Once all the kernel parameters for each kernel are extracted, the frame level image processing application is converted to block level application kernel in a sequential manner. This is followed by validation for bit exactness for each kernel. Optionally the user can also add these parameters are pragmas to assist the code parser.

3.2. Local Optimization – One Kernel Approach

The simplest approach to convert a frame based image processing application to a block based application is to convert each kernel one at a time without chaining the consecutive kernels together. This approach does not have any data dependency acros s multiple kernels in the application. We call this approach as the local optimization approach where the performance is optimized locally specific to individual kernels.

To aid the understanding of the discussion in the next few sections, let us first define two terminologies namely compute bound and data bound. A particular kernel is considered to be compute bound if the processing time on the CPU is much larger compared to the data transfer times of the DMA engine to bring data from external DDR memory into internal CPU memory (L1/L2). Otherwise, the kernel is considered to I/O bound. The primary goal during the optimization of any image processing application is to make the application compute bound.

Given the kernel and the relevant kernel parameters such as input and output buffer sizes, kernel block size, block based characteristic, we first try to compute the remaining memory parameters namely compute time and data transfer time that is very crucial to achieve the compute bound criteria for the kernel under consideration. The compute time for a kernel can be calculated using a cycle accurate device simulator that models without any memory overheads for that kernel. The data transfer times can be inferred from the platform support package and the buffer sizes. To provide a simple example, let us consider a simple Sobel 3x3 edge filter kernel. The input buffer size is say 640x480 and the output buffer size is 640x478. This means that given 10 lines of input, the filter generated 8 lines of output with a shrinking factor of 2. The shrinking factor gives the difference between the input and the output lines. If the time taken to process these 10 lines input is say 100 cycles and the time taken for the DMA to copy 10 lines of data from external DDR to internal L1/L2 is 200 cycles, then the operation is I/O bound. Hence the compiler parser reiterates by bringing in a larger block of input data (say 20 lines). The compute time for this block of 20 lines to generate 18 lines of output is say 150 cycles and the data transfer time for 20 lines of input is say 250 cycles, then unfortunately the kernel is still I/O bound. The code parser again reiterates with larger blocks of data in steps to converge on a solution such that the kernel under consideration is compute bound.

Finally, the each frame level kernel is replaced with an equivalent block level kernel. The input image is split into smaller block of data, the optimum size of which is calculated at convergence to achieve compute bound operation. Block accesses are performed by the CPU using memory copy (memcpy) function to copy data from external memory to internal memory. Later the memcpy functions are replaced by equivalent DMA APIs with ping/pong buffering. The local optimization is summarized in Figure 4 in which the chaining of kernels (Global optimization) is absent. The trivial extension to this local optimization approach is to perform global optimization of all kernels present in the application. This is discussed in the next section.

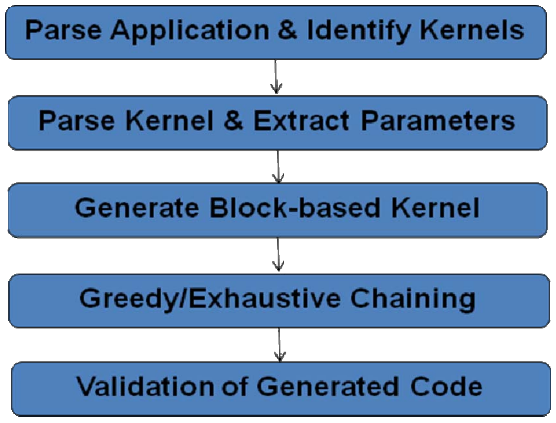

Parse Application & Identify Kernels
Parse Kernel & Extract Parameters
Generate Block-based Kernel
Greedy/Exhaustive Chaining
Validation of Generated Code

Figure 4: High Level Block Diagram of the code generation process

3.3. Global Optimization – Chaining of Kernels

Some frameworks try to reconstruct a dependency graph across kernels and build a graph and then exploit the parallelism. These approaches are useful for heterogeneous multi-core parallelization, but the same has limitations for single core algorithm acceleration. The implementation explicitly defines a linear sequence of kernels and chaining is defined as the partition of the whole set into subsets which will be executed with one DMA transfer.

A greedy approach is used to chain several kernels especially the ones which are bottlenecked by data-transfer. Finally the set of kernels is split into ordered disjoint sets which would be executed together on one access to the memory. As this process is offline and the set of kernels is typically small, finding an optimal partition using all possible combinations is feasible.

The algorithm to chain kernels is explained below. Because of the offline nature of the problem and small cardinality of kernels, a brute force approach is followed. The complete methodology of code generation process is summarized in Figure 4 and the flow graph of the algorithm is shown in Figure 5.

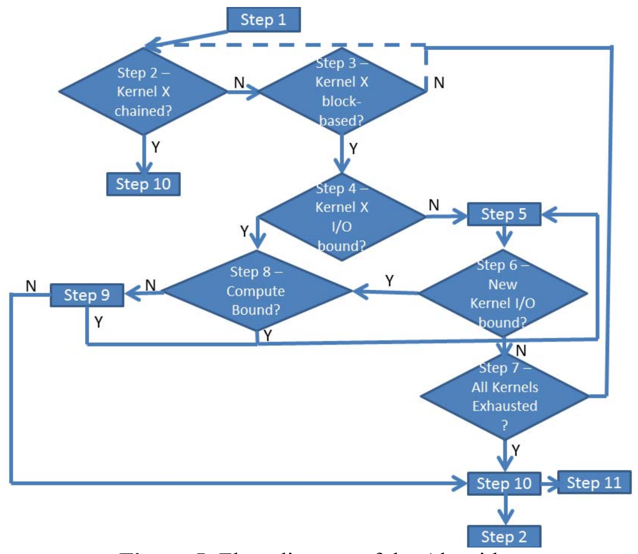
Algorithm:

Step 1: Determine Kernel parameters and memory parameters for each kernel in the application. This is obtained from the code parser and the cycle accurate simulator

Step 2: If the kernel *x* already *chained* go to Step 10 otherwise go to Step 3

Step 3: If the kernel *x* is *block based*, then go to Step 4, otherwise disable *chaining* of this kernel and go to Step 2 for testing the next kernel

Step 4: If the kernel *x* is *I/O bound*, then go to Step 8, otherwise go to Step 5



Step 1
Step 2 – Kernel X chained?
Step 3 – Kernel X block-based?
Step 4 – Kernel X I/O bound?
Step 5
Step 6 – New Kernel I/O bound?
Step 7 – All Kernels Exhausted ?
Step 8 – Compute Bound?
Step 9
Step 10
Step 11
Step 2
N
Y

Figure 5: Flow diagram of the Algorithm

Step 5: Chain kernels *x* and *x-1* and name the new kernel as *x* and re-compute the kernel parameters and memory parameters

Step 6: If the new kernel *x* is *I/O bound* then go to Step 8, otherwise go to Step 7

Step 7: If all the kernels are exhausted, go to Step 10, otherwise go to Step 2

Step 8: Chain kernels *x* and *x-1* and re-compute all parameters and determine whether this new *chained* kernel is compute bound or *I/O bound*. If *compute bound*, go to Step 5 for testing next kernel, otherwise go to Step 9

Step 9: Chain kernels *x* and *x+1* and recompute all parameters and determine whether this new chained kernel is *compute bound* or *I/O bound*. If *compute bound*, go to Step 5 for testing next kernel, otherwise go to Step 10

Step 10: Repeat chaining of successive kernels until all the kernels are exhausted. If a schedule is obtained, then update the parameters and go to Step 2 for testing the remaining kernels, otherwise go to Step 11

Step 11: Validate the generated code for bit exactness and publish results

3.4. Determining Buffer Sizes

Consider ***K*** kernels with shrinking factors $S = \{s_1, s_2, s_3 .... s_k\}$, where s_i is a positive integer. The shrinking factor is the difference between the number of lines on the input and the number of lines on the output. The shrinking factor aids in data reuse across multiple iterations of the kernels, thereby reducing the amount of data needed to be transferred from external memory to internal memory. Let the number of lines in the internal local memory be ***M***, where each line can be considered to be of ***N*** pixels length. ***M*** can be calculated as $M = \sum_{i=1}^{k} s_i$.

Let the kernel computation time of ***K*** kernels be $C = \{c_1, c_2, c_3 .... c_k\}$, and the data transfer times for these kernels be $D = \{d_1, d_2, d_3 .... d_k\}$. Hence, a kernel k_1 is chained with kernel k_2 when $c_1 < d_1$ and $(c_1 + c_2) < (d_1 + d_2)$, under the condition that chaining k_2 and k_3 is not providing significant improvement. Once the kernels k_1 and k_2 are chained together, k_1 and k_2 are treated as a single kernel and the data transfer time is reduced from (d_1 + d_2) to d_1 and the same algorithm is repeated for the remaining kernels. This way an optimal chaining is reached to achieve best possible performance. More details of the algorithm are discussed in Algorithm 1 and the block diagram is provided in Figure 4. The number of lines *L* that can be processed in one iteration when kernels $\{k_1, k_2, k_3 .... k_k\}$ are chained together is calculated using the following relation.

$$\sum_{i=1}^{k} (w_i l_i + L) + 2 * (L + 1) = I - S \qquad (1)$$

where w_i = weighting factor that depends on element size,
l_i = kernel block size of kernel *i,*
L = number of minimum lines that should be processed to achieve maximum optimization
I = available internal memory on *CPU (L1/L2)* in bytes
S = scratch pad memory required for state variables, look up tables, etc in bytes

4. Case Study – Canny Edge Detection

Canny edge detection comprises of 4 steps namely Gaussian smoothing, gradient filtering, non-maxima suppression and edge relaxation. The tool determines that edge relaxation is not a block-based function and is ignored. Appropriate kernel parameters from the other three functions are parsed and suitable buffers are allocated as shown in Figure 6. As part of Prolog all the listed buffers are filled with the output data and in steady state, each step provides equal number of output lines. As illustrated in Figure 5, we show one line of output being produced. From the seven lines of input, we obtain five lines of Gaussian smoothed output, three lines of Gradient filtered output and one line of non-maximum suppressed output.

For the computation of a new output line, we discard the first line, re-use the remaining lines and bring in a new line. This can be efficiently implemented using a circular buffer. Some processors support circular addressing in HW, otherwise it can be implemented easily using modulo arithmetic addressing. Ping-pong buffers are used only for the first input buffer and the las t output buffer.

Based on the equation 1, the number of lines to be brought into internal memory is calculated. Once this is calculated, the appropriate buffers are allocated in the internal memory and an equivalent code for the chained block-based version is generated.

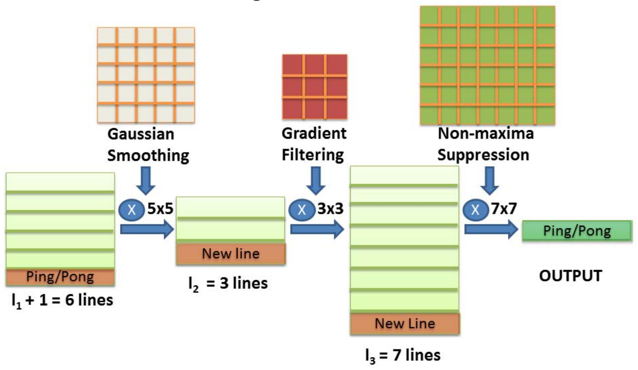


Figure 5: Buffer Allocation

Application/Function	Conventional (cycles/ pixel)	Proposed (cycles/ pixel)	Speed Up
Gaussian Filter 7x7	6.55	2.2	2.97
Gradient and Mag	3.4	1.0	3.4
NMS 7x7	7.2	2.4	3.0
Double Thresholding	8.5	3.0	2.83
Edge Relaxation	3.0	3.0	1.0
Sobel Filter 3x3	4.5	1.7	2.6
Canny Edge Detection	84	35	2.4
Lane Departure Warning	31.5	13.2	2.38

Table 1: Performance comparison between cache based mode and DMA based model

5. Results and Discussion

We provide the comparison of the performance benchmarks with and without use of our proposed algorithm. Performance improvement depends on the compute cycles and number of input/output buffers. Larger the number of buffers larger the cache misses and memory overheads and hence better the performance improvement. Table 1 below summarizes the performance speedup. It is interesting to note that the Edge relaxation function cannot be done using block based method. Hence there is no improvement obtained for this kernel. The implementation of these algorithms are based on the DSP optimized Vision Library VLIB [7].

The current implementation of the parser handles C code only. It also assumes certain structure in the implementation so that it is easier to parse and identify the kernels. One such assumption is that each kernel should be a separate function or an easily identifiable loop segment. These challenges are related to compiler and parser technologies which are beyond the scope of this paper. The proposed algorithm was able to handle most of the low-level computer vision algorithms [4]. There is also limited support for connectivity-based image processing algorithms [5]. There were instances of failures due to the nature of the implementation and limitation of the current implementation of the parser. This was caught in the verification stage and the original code was restored.

6. Conclusion

In this paper, we discussed a novel automated parser which generates memory efficient block based code from a frame based code at compile time. The performance improvement achieved using this scheme is of the order of *~2-4X*, and the power dissipation is also reduced by about *40-50* percent. The proposed scheme does not require any code involvement of the programmer since every memory optimization is performed by the proposed solution automatically. This tool is expected to provide a significant performance improvement irrespective of the platform that is being used to develop applications. Our method addresses typical vision/image processing algorithms. As part of future work, a parser for linear connectivity based algorithms will be dealt with.

References

[1] Ko, D. I., & Bhattacharyya, S. S. Modeling of block-based DSP systems. The Journal of VLSI Signal Processing, 40(3), 289-299, 2005.

[2] TMS320C6000 DSP Cache User's Guide. TI Technical Report, SPRU656A, 2003.

[3] Data Parallelizing Framework Reference Guide. Uncanny Vision Documentation, 2013.

[4] Kisacanin, B, Examples of low-level computer vision on media processors. In Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on (pp. 135-135). IEEE.

[5] Kiran, B. R., Anoop, K. P., & Kumar, Y. S. Parallelizing connectivity-based image processing operators in a multi-core environment. In Communications and Signal Processing (ICCSP), 2011 International Conference on (pp. 221-223). IEEE.

[6] S.K. Yogamani. A Tutorial on Optimizing Vision Algorithms on TI DSPs. Texas Instruments Application Report, SPNA165, 2012.

[7] Vision Library API Reference Guide, Texas Instruments Documentation, 2012.

[8] Ko, D. I., Won, N., & Bhattacharyya, S. S. Buffer management for multi-application image processing on multi-core platforms: Analysis and case study. In Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on (pp. 1662-1665). IEEE.

[9] UncannyDP: Data Parallelizing Framework Reference Guide. Uncanny Vision Documentation, 2013 http://www.uncannyvision.com/uncannydp/