Stream Image Processing on a Dual-Core Embedded System

Michael G. Benjamin and David Kaeli

Northeastern University, Computer Architecture Research Laboratory
409 Dana Research Center, 360 Huntington Ave, Boston, MA 02115, USA
{mbenjami,kaeli}@ece.neu.edu

Abstract. Effective memory utilization is critical to reap the benefits of the multi-core processors emerging on embedded systems. In this paper we explore the use of a stream model to effectively utilize memory hierarchies. We target image processing algorithms running on the Analog Devices Blackfin BF561 fixed-point, dual-core DSP. Using optimized assembly to effectively use cores reduces runtime, but also underscores the need to mitigate the memory bottleneck. Like other embedded processors, the Blackfin BF561 has L2 SRAM available. Applying the stream model allows us to effectively make full use of both cores and the L2 SRAM. We achieve almost a 10X speedup in execution time compared to non-optimized C code.

1 Introduction

Convergent, embedded architectures combine multiple instruction sets and allow traditional digital-signal processing (DSP) platforms to also run micro-controller (MCU) code. To develop applications for these platforms, developers should try to leverage open source image and video processing libraries to drastically reduce development time of embedded computer vision applications. By utilizing DSP and other special-purpose hardware available on the processor, the performance of these applications can be greatly improved. But with highly compute-optimized code, the memory bottleneck becomes even more apparent.

The growing gap between processor and memory speeds has long been acknowledged [1] and has been attacked using on-chip caches. But image and video applications exhibit characteristics not well-suited for conventional caches. Images have 2D spatial locality, while caches capture only one dimension [2,3]. Image processing shows little temporal data reuse, causing cache pollution. Cache lines corresponding to processed data are not reused and must be replaced often. This characteristic of image-based processing need cache structures that can accomodate both types of locality [4,5]. An alternative use of the on-chip memory is as scratch-pad memory, with data communication controlled explicitly by the programmer [6].

The stream model of computation has recently been proposed to address memory bottlenecks. In short, the stream paradigm decouples computation and memory accesses to ensure parallelism and locality. We examine the use of the stream paradigm in conjunction with assembly optimizations on a representative set of image processing algorithms, running on the Blackfin embedded processor from Analog Devices, Inc. (ADI) - a good example of a convergent architecture.

S. Vassiliadis et al. (Eds.): SAMOS 2007, LNCS 4599, pp. 149–158, 2007.

Section 2 discusses the stream model in further detail. Section 3 discusses the convergent architecture of the dual-core Blackfin BF561 processor. Section 4 describes the main target application discussed in this paper, an edge detection program. We also discuss our scheme to produce optimized execution on the convergent Blackfin DSP architecture. Section 5 presents our results and section 6 concludes the paper.

2 Stream Computing Paradigm

The goal of the stream paradigm is to maximize data locality while exposing parallelism. Stream applications decouple two aspects of computing: performing computations, and data communication required to feed the computations. Streams are sets of data elements to be processed by compute kernels, which are sequences of instructions applied to each element in a stream.

Locality can be maximized because during a kernel's execution, all data accesses can be served by local memory storage (maximizing kernel locality) and because results produced by one kernel are quickly consumed by the next (maximizing producer-consumer locality).

Given the necessary computational resources, multiple kernels can operate simultaneously in a pipeline, exposing thread-level parallelism. For each thread and within a kernel, independent instructions can operate at the same time, exposing instruction-level parallelism. A particular instruction could be vectorized and applied to many elements of the data stream using SIMD hardware, exposing data-level parallelism [7,8].

2.1 Stream Applications, Languages, and Compilers

To quickly process high definition images or video sequences, data should be locally available to computational units. For large images, a common approach is to partition the data, effectively maintain intermediate results in low-latency memory, and continuously process the resulting subimages. This fundamental streaming nature is apparent in a number of applications and modern graphics processing units (GPUs) are designed around a stream model both in their hardware [9] and software [10] implementations. A number of languages and compilers have been developed including StreamIT [11], Stream-C and Kernel-C [12].

2.2 Stream Architectures

The rising demand for media processing has motivated the development of a number of architectures and re-examination of existing architectures to work with a stream model. Processors like Imagine [13,14], Merrimac [15], and Stream Processors Inc.'s Storm-1 [16] have been developed to exploit the stream model. To support diverse, dynamic applications, architectures that can be reconfigured to execute efficiently for many classes of applications have been introduced. Architectures such as TRIPS [17] and RAW [18] can be described as polymorphous - that is, they can morph between a number of operating modes, each capturing some class of applications.

Researchers have even explored using the stream paradigm on existing general-purpose processors [19]. By using L2 cache as local storage, and multithreading for parallelism, scientific applications can achieve moderate speedups (27%). But cache overhead limits the effectiveness of this approach. For architectures which support L2 SRAM acting more like scratchpad memory (common in many embedded processors, e.g., the ADI Blackfin), there is potential for higher performance.

3 Blackfin Processor

The Blackfin is an embedded architecture based on the MicroSignal Architecture developed jointly by ADI and Intel [20]. It supports 8, 16, and 32-bit arithmetic operations, but is optimized for 16-bit operations. The Blackfin reduces power and part cost by integrating RISC and DSP processor capabilities into a single chip. But rather than fusing two cores together, the Blackfin ISA was designed from the beginning with this duality in mind. The 32-bit RISC instruction set is variable-length and supports Single Instruction, Multiple Data (SIMD) operations. As shown in Fig 1, the DSP instruction set makes use of dual 16-bit multiply-accumulate (MAC) units. Additionally, a video pixel instruction set supports four 8-bit video ALUs which can be utilized with vector instructions for addition, multiplication, averaging, and sum-of-absolute-differences calculations [21].

The BF561 processor is a Blackfin derivative with two cores, clocked at up to 600 MHz. The BF561 has 128 KB on-chip L2 SRAM, clocked at 300 MHz, and 100 KB in-core L1 SRAM, which is split between instruction, data, and scratch-pad memories.

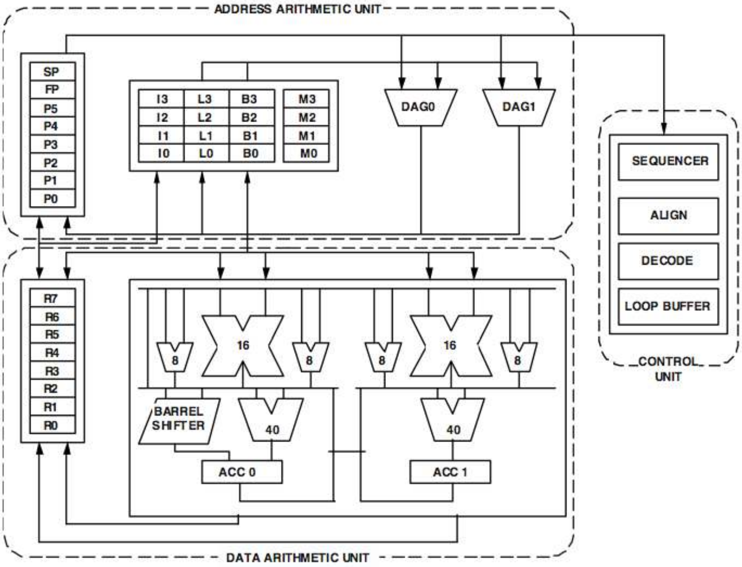

Fig. 1. The ADI Blackfin processor core

L1 can be configured as either SRAM or cache. The BF561's L2 memory can not be configured as cache [22], but the programmer is able to map data into regions in L2. Without the turn-key solution of using L2 as cache and given the limitations of cache for image and video processing discussed in the introduction, we examine the use of L2 as a local store in the stream paradigm.

4 Implementation

Many image processing algorithms are made up of a sequence of convolutions, which transform images using a set of kernel matrices. To avoid confusion with the computational kernels used in the stream model, we refer to kernel matrices as "masks." For example, an image represented by the 2D matrix $\mathbf{F} := (f_{x,y})_{m\times n}$ can be transformed into another image represented by the matrix $\mathbf{G}$ by convolving $\mathbf{F}$ with a mask, denoted $\mathbf{H}$, by the 2D convolution equation:

$$g_{x,y} = \sum_{i=-1}^{1}\sum_{j=-1}^{1} h_{j+2,i+2} \times f_{x-j,y-i}$$

Such convolutions are frequently used to update pixel values based on neighborhood operations. For example, for images which contain noise, averaging a pixel based on its neighborhood can reduce the intensity of the noise, thus smoothing or blurring the image. In order to determine edges in an image, approximations of the first and second derivative can be used as masks to determine if the intensity level for a pixel neighborhood is uniform or changes (implying an edge). A Gaussian mask implements noise reduction; two Sobel masks implement horizontal and vertical edge detection:

$$\mathbf{Gauss} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \mathbf{Sobel_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \mathbf{Sobel_y} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

In general such image convolution algorithms can be described as follows, where mem_i and mem_o are data sets in memory for input data and processed output data, respectively:

```
IMAGEPROCKERNEL ($mem_i, m, n, mask, mem_o$)
 for $y \leftarrow 0$ to $m-1$
   do for $x \leftarrow 0$ to $n-1$
     do $p$ = GETPIXEL($x, y, mem_i$)
     2D CONVOLVE( $p, mask$ )
     SETPIXEL($p, y, x, mem_o$)
```

The control flow of an image processing program can be described as a sequence of kernels (such as the IMAGEPROCKERNEL algorithm). To process an image, each $kernel_i$ would execute, convolving every pixel p in the $m \times n$ source image src with the convolution mask(s), and setting appropriate pixel values in the destination image dst. Performance can be increased by ensuring calls to GETPIXEL and SETPIXEL access low-latency memory.

```
ProcessImage
foreach $kernel_i$ in Control Flow
    Run($kernel_i$ $(src, m, n, mask, dst)$)
```

We examine an edge detection program using a Gaussian blur convolution to reduce noise and a Sobel gradient-operator convolution to highlight both horizontal and vertical edges in a high-definition image. The resolution used is 1920x1080 (WxH) which requires 2,073,600 pixels to be convolved. For a 3x3 mask, this requires 9 multiplications and 8 additions per pixel. Therefore, a single convolution would require about 18.7 MMACs. Our program uses three such convolutions, and therefore the ideal is about 56 MMACs.

The actual benchmark code used is taken from existing codebases [23] and from the ADI Blackfin SDK [24]. We examine the assembly-level optimizations given in the SDK code to reduce the execution time on each core, and then attempt to optimize the use of both cores to reduce overall execution time. The computational optimizations underscore the need for efficient memory usage.

4.1 Assembly Optimizations

As discussed in section 3, the Blackfin has a dual-MAC architecture: to carry out convolutions, two pixels can be convolved at once. By pushing the unique values of a mask onto the stack and popping them when needed, the program can quickly fetch $h_{x,y}$ mask values. Using parallel-issue instructions, the Blackfin can execute two MAC operations to set two $g_{x,y}$ pixel values while fetching the next $f_{x,y}$ pixel and $h_{x,y}$ mask values in the same clock cycle. This technique allows the 56 MMACS to require only 28 Mcycles, which running on a 600 MHz core clock is less than 0.05 sec. This runtime is the theoretical lower limit for the convolutions alone, and the delay added due to memory latency is substantial - emphasizing the need for efficient memory use.

4.2 Dual-Core Utilization

A natural approach to utilize symmetric cores is to divide the image into partitions, giving each core its own portion of the data. This can be described as partitioning the data-flow of the program and has also ben called the "homogenous model" [25]. Another approach is to partition the compute kernels and give each core some set of kernels to perform; the resulting data would flow from one core into the next. This approach can be described as partitioning the control-flow of the program, and embodies some of the characteristics of the stream model.

In order to add more kernels to the stream, we utilize C data structures representing kernels, which include pointers to the input/output records and a callback for the kernel's function, and the three routines discussed below:

- addKernel - adds a kernel to a linked list representing the program's control flow
- newStreamRec - allocates memory for input/output records of a kernel
- mapStream - maps existing records for a kernel

At initialization, we use the above routines to set up the control flow of a program. Following initialization, we call each kernel as specified. To implement other image processing programs using our method, the addition of new kernels or stream mappings only requires a new initialization for each core, and does not require a specialized stream compiler.

To use the two symmetric cores on the BF561, we examined both approaches discussed above. For data-flow partitioning, each core processes half the image. For control-flow partitioning, each core processes half of the computational kernels. This approach is viable for our two convolution kernels, but for programs that lack a balanced set of computational kernels, a heuristic approach may be needed.

4.3 Memory Hierarchy Utilization

In order to make use of the BF561's L2 SRAM as local storage memory, we examined two approaches. The first uses regions of L2 SRAM on a per-kernel basis. It copies subimages into input records associated with a computational kernel (denoted $kernel_i.in$), runs the kernel, saves results into output records (denoted $kernel_i.out$), and saves these records back to SDRAM after computation completes; this process is then repeated for the next kernel in the control flow of the program. A side-effect of this approach is that the results of each kernel can be saved, but at the cost of increased memory access for each kernel. This approach is summarized in the following pseudocode:

```
PER-KERNEL CONVOLUTION
foreach $kernel_i$ in Control Flow
    do foreach $SUBIMAGE$ in $IMAGE$
        do COPY($SUBIMAGE, kernel_i.in$)
        RUN ($kernel_i(kernel_i.in, m, n, kernel_i.out)$)
        COPY($kernel_i.out, SUBIMAGE$)
```

The second approach uses the stream processing paradigm. Under this model, SDRAM is only accessed for compulsory reads or completion writes (i.e., only after all k kernels in the control flow have been processed). This approach is summarized in the following pseudocode:

```
STREAM CONVOLUTION
foreach $SUBIMAGE$ in $IMAGE$
    do COPY($SUBIMAGE, kernel_1.in$)
    foreach $kernel_i$ in Control Flow
        do RUN($kernel_i(kernel_i.in, m, n, kernel_{i+1}.in)$)
    COPY($kernel_k.out, SUBIMAGE$)
```

Unlike the previous approach, the results of each kernel are not saved - only the final results after every kernel has been run is saved to SDRAM. But for programs where the intermediate results are not important, reducing the number of main memory accesses provides higher performance.

5 Results

We produce the following results on a live Analog Devices BF561 system. Figures 2 and 3 show the results using hardware counters.

Figure 2 shows how the different methods reduce the cycle count. For the single-core (SC) case, the use of L2 in the partitioned data-flow method reduces the runtime by nearly half. The use of a second core is effective only when the cores are not waiting for data (i.e., when data is stored in on-chip memory). Ideally, using the second core would half the runtime, but when SDRAM is used the additional core only provides a 27% increase in performance. Only when the latency penalty of SDRAM access is minimized by using L2 do we achieve a 2X speedup due to the second core.

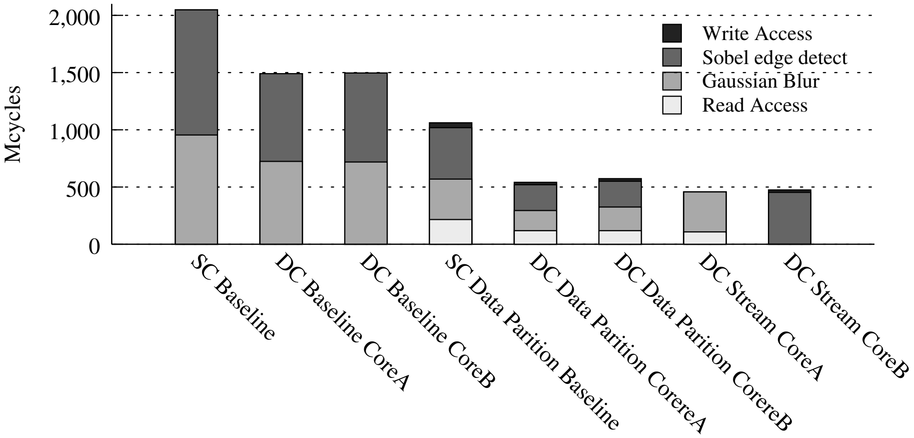

Fig. 2. C implementations for single-core (SC) and dual-core (DC) approaches

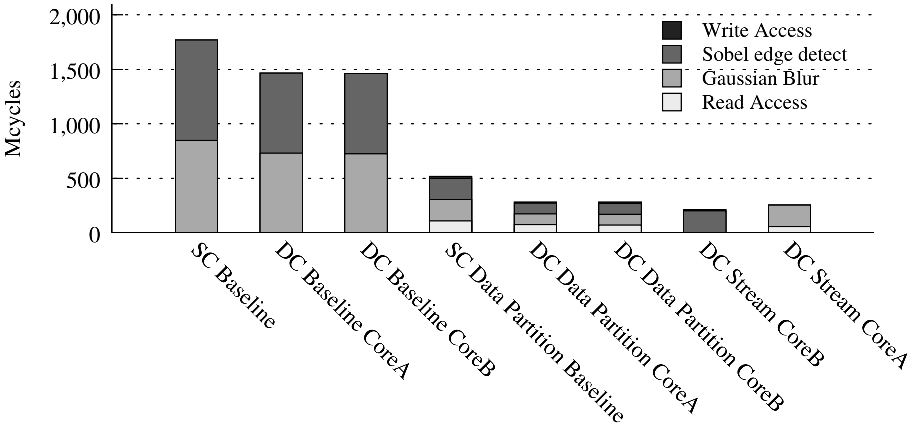

Fig. 3. Assembly implementation of the 3x3 convolution program

Table 1. Runtimes using single and dual-core, using different memory management approaches, and comparing C and ASM implementations

	C Runtime (Speedup) in secs.	ASM Runtime (Speedup) in secs.
Single-core baseline approach	3.99 (1.0)	3.64 (1.10)
Dual-core split data	3.15 (1.27)	2.45 (1.63)
Single-core with L2	1.80 (2.22)	1.08 (3.69)
Dual-core split data with L2	0.92 (4.34)	0.44 (9.07)
Dual-core streamed approach	0.80 (4.99)	0.42 (9.50)

Figure 3 shows that even though optimized assembly is being used instead of C, the performance increase is only 10% for a single-core and 63% for a dual-core. Our image processing application is truly limited by the SDRAM's memory latency.

Using L2 to alleviate this bottleneck exposes the benefits of using assembly. The performance speedup is 1.7X when using assembly for the case of a single-core with L2 - nearly double, as is expected due to the effective use of both MAC units. The same is true for both dual-core cases as well.

Results from hardware cycle counters are converted to runtimes and given in Table 1, along with the speedup compared to the single-core baseline. Due to the limiting effects of memory latency, the streaming paradigm performs better in both C and assembly cases. The stream approach only accesses SDRAM on compulsory accesses and upon completion of the convolutions. We see that we are far from the ideal runtime of 0.05 sec because L2 operates at 300 MHz and the operations to manage dataflow dominate performance.

6 Conclusion

The Blackfin BF561 is a dual-core DSP/MCU convergent architecture with a 128 KB L2 SRAM. For image and video processing applications, we demonstrate that the stream computing paradigm provides an effective model for making use of the computational resources and L2 memory of the BF561. Using assembly code that utilizes dual-MAC hardware doubles the performance but only when most of the memory accesses are to L2 SRAM. Utilizing the stream model to limit SRAM accesses provides the best performance.

Acknowledgment

The authors would like to thank Mimi Pichey, Giuseppe Olivadoti, Richard Gentile, and Ken Butler from Analog Devices for their generous donation of equipment and support of this work. We also extend our gratitude to the anonymous reviewers for their comments and suggestions.

References

1. Wulf, W.A., McKee, S.A.: Hitting the Memory Wall: Implications of the Obvious. SIGARCH Computer Architecture News 23, 20–24 (1995)
2. Cucchiara, R., Massimo Piccardi, A.P.: Exploiting Cache in Multimedia. In: Proc. of Int'l Conference on Multimedia Computing and Systems, vol. 1, pp. 345–350 (1999)
3. Pati, A.: Exploring Multimedia Applications Locality to Improve Cache Performance. In: Proc. of 8th Int'l Conference on Multimedia, pp. 509–510 (2000)
4. Naz, A., Kavi, K., Sweany, P., Rezaei, M.: A Study of Separate Array and Scalar Caches. In: Proc. of the 18th Int'l Symposium on High Performance Computing Systems and Applications, pp. 157–164 (2004)
5. Naz, A., Rezaei, M., Kavi, K., Sweany, P.: Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers. In: Proc. of the 2004 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture, pp. 41–48 (2004)
6. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In: Proc. of the 10th Int'l Symposium on Hardware/Software Codesign, pp. 73–78 (2002)
7. Dally, W.J., Kapasi, U.J., Khailany, B., Ahn, J.H., Das, A.: Stream Processors: Programmability and Efficiency. ACM Queue 2, 52–52 (2004)
8. Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ahn, J.H., Mattso, P., Owen, J.D.: Programmable Stream Processors. ACM Computer 8, 54–62 (2003)
9. Venkatasubramanian, S.: The Graphics Card as a Stream Computer. In: Workshop on Management and Processing of Data Streams (2003)
10. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transactions on Graphics 23, 777–786 (2004)
11. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., Meli, A.S., Lamb, A.A., Leger, C., Wong, J., Hoffmann, H., Maze, D., Amarasinghe, S.: A Stream Compiler for Communication-Exposed Architectures. SIGPLAN Not. 10, 291–303 (2002)
12. Mattson, P.: A Programming System for the Imagine Media Processor. PhD thesis, Stanford University (2001)
13. Rixner, S., Dally, W.J., Kapasi, U.J., Khailany, B., Lopez-Lagunas, A., Mattson, P.R., Owens, J.D.: A Bandwidth-Efficient Architecture for Media Processing. In: Proc. of the 31th Int'l Symposium on Microarchitecture, pp. 3–13 (1998)
14. Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., Chang, A., Rixner, S.: Imagine: Media Processing with Streams. IEEE Micro 21, 35–46 (2001)
15. Dally, W.J.: Merrimac: Supercomputing with Streams. In: Proc. of the Conference on Supercomputing (2003)
16. Stream Processing: Enabling a New Class of Easy to Use, High-Performance Parallel DSPs. White Paper 1.9, Stream Processors Inc. 455 DeGuigne Drive Sunnyvale, CA 94085, USA (2007)
17. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Ranganathan, N., Burger, D., Keckler, S.W., McDonald, R.G., Moore, C.R.: TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. ACM Transactions on Architecture and Code Optimization 1, 62–93 (2004)
18. Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A.: Baring It All to Software: RAW Machines. Computer 30, 86–93 (1997)

19. Gummaraju, J., Rosenblum, M.: Stream Programming on General-Purpose Processors. In: Proc. of the 38th Int'l Symposium on Microarchitecture, Washington, DC, USA, pp. 343–354. IEEE Computer Society Press, Los Alamitos (2005)
20. Kolagotla, R.K., Fridman, J., Aldrich, B.C., Hoffman, M.M., Anderson, W.C., Allen, M.S., Witt, D.B., Dunton, R.R., Booth, L.A.J: High Performance Dual-MAC DSP Architecture. IEEE Signal Processing 19, 42–43 (2002)
21. Analog Devices, Inc. One Technology Way, Norwood, MA 02062, USA: ADSP-BF53x/BF56x Blackfin Processor Programming Reference. 1.0 edn. (2005)
22. Analog Devices, Inc. One Technology Way, Norwood, MA 02062, USA: ADSP-BF561 Blackfin Processor Hardware Reference. 1.0 edn. (2005)
23. Green, B.: Edge Detection Tutorial (2002), http://www.pages.drexel.edu/~weg22/edge.html
24. Analog.com: Software Development Kit (SDK) Downloads (2007), http://www.analog.com/processors/platforms/sdk.html
25. Ning, K., Yi, G., Gentile, R.: Single-chip Dual-core Embedded Programming Models for Multimedia Applications (2005), http://www.ecnmag.com/article/CA502854.html