Use of an Embedded Configurable Memory for Stream Image Processing

Michael G. Benjamin and David Kaeli
Northeastern University Computer Architecture Research Group
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
{mbenjami, kaeli}@ece.neu.edu

***Abstract*— We examine the use of the embedded Blackfin BF561 processor for high-definition image processing using the stream model of computing. The Blackfin features a configurable memory hierarchy that minimizes the Memory Wall effect. We describe the stream model and its application to the BF561 to utilize low-latency on-chip memory and compare to a worst-cast baseline using SDRAM only. We find a 2X to 3X speedup in the edge detection of a 1920x1080 pixel image using C and 3X to 11X speedup using assembly.**

I. Introduction

As high definition (HD) cameras and displays become commonplace, the need to quickly process large images increases. The embedded processors found in such devices have, for the most part, not followed the trend of increasingly faster clock speeds seen in general-purpose processors. High clock speeds translate to higher power usage, which in turn requires cooling systems to maintain reliable performance. Such requirements are ill-suited to the constrained environments of embedded systems. For this reason, and because embedded processors tend to be geared more to specific classes of applications, embedded chips have specialized hardware resources to do more per cycle, rather than reducing the cycle time.

Many image processing routines can be expressed as a set of instructions, called compute kernels, which transform raw image data into meaningful information. Vector processing units, special-purpose hardware such as video ALUs (vALUs), and convergent cores have extended some embedded architectures to allow multiple pixels to be transformed at once. But because of the vast amounts of data needing to be processed, memory performance is critical. Researchers have examined the memory hierarchy and proposed methods to minimize the latency associated with accessing high-capacity, slow off-chip SDRAM (called the Memory Wall effect [1], [2]). This need to feed the computational units which may otherwise be starved for data motivates our work and our examination of the stream model of computing.

In this paper, we examine two approaches to using low-latency L2 and L1 SRAM available in the memory hierarchy of the Analog Devices Inc (ADI) Blackfin BF561 embedded processor. The first uses SRAM to accelerate each kernel individually, and the second applies the stream model which uses SRAM for inter-kernel dataflow. We compare these approaches to a baseline of only using SDRAM for edge-detection in a HD image.

Section II discusses generic image processing algorithms and memory considerations. Section III explains the stream paradigm further. Section IV describes the ADI Blackfin embedded processor. Section V describes the two approaches for low-latency memory usage. Section VI describes the workload examined here. Section VII describes the implementation of an edge-detector using the BF561 and discusses results. Section VIII discusses the related work for utilizing configurable memory hierarchies and applying the stream model to existing architectures. Section IX concludes the paper. Section X outlines future work.

II. Image Processing and Memory

Many image processing routines can be summarized by the algorithm below, where mem_i and mem_o are memory locations; $rows$ and $cols$ are the number of rows and columns in the image, respectively:

```
ImageProcKernel (mem_i, rows, cols, mem_o)
1  for y ← 0 to rows − 1
2   do for x ← 0 to cols − 1
3     do p = GetPixel(x, y, mem_i)
4     Transform (p)
5     SetPixel(p, x, y, mem_o)
```

A program whose control flow consists of a sequence of such transformation kernels can itself be described with the following algorithm:

```
ProcessImage
1  foreach kernel_i in Control Flow
2    Run(kernel_i(origImage, rows, cols, procImage))
```

The Run routine simply runs the $kernel_i$ algorithm and transforms every pixel in *origImage* into the appropriate pixel in *procImage*. Each kernel's Transform routine can be optimized using SIMD instructions that make use of replicated hardware and media extensions to the architecture. But the GetPixel and SetPixel routines access memory structures

which often are located in off-chip SDRAM (especially for HD images which are much larger than many on-chip memories).

Better overall performance can be achieved by having needed data in low-latency, on-chip memory. Conventionally, caching has been used to move data into SRAM, but the usual cache benefits are limited for image processing. Images have two dimensional spatial locality, but caches only capture 1D locality [3], [4]. Also, due to its nature, image processing has limited temporal locality - this lack of data reuse can cause cache pollution.

Mechanisms at both the architecture and application have been developed to deal with this issue. Architecturally, split caches, stream buffers, and adaptive caches [5], [6], [7], [8] have been introduced to utilize both spatial and temporal locality. Scratch-pad memories have been presented as a cache alternative [9]. From the application perspective, frameworks and methodologies have been developed to increase data locality [10] and to perform source-to-source transformations to ensure [11] data is consumed soon after it is produced. Novel architectures have also been introduced to offload processing to the memory elements themselves such as vectorized intelligent RAM, processor-in-memory, etc. [12], [13].

An alternative approach has been described in the context of stream processors. Memory access to SDRAM only happens initially to get data, when it is compulsory to do so, and after the completion of all transformation kernels, with inter-kernel communication via on-chip local stores. The next section describes the stream model in detail.

III. Stream Computing Paradigm

The stream model of computing seeks to maximize locality while exposing parallelism [14]. Stream computing decouples memory and computation into streams and kernels, respectively. Streams consist of a set of data records which are to be processed. Kernels describe a sequence of instructions that are applied to each input record and whose results are saved into output records. A stream program is expressed as data streams which pass through a sequence of computational kernels.

Figure 1 shows an example of the dataflow of a generalized stream program. For example, many image processing programs can be described using a stream of pixel blocks and a set of n transformation kernels.

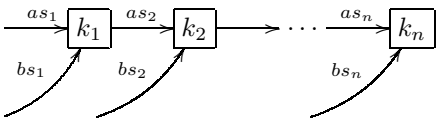

Fig. 1. An example dataflow graph for a generic stream program.

Locality is maximized during a kernel's execution, because all data accesses can be served by a local memory store (kernel locality) and because results produced by one kernel are quickly consumed by the next (producer-consumer locality). Given the 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 [15].

The rising demand for media processing has motivated the development of a number of architectures and re-examination of existing architectures to work with the stream model. Architectures such as Imagine [16], [17], Merrimac [18], and Stream Processors Inc.'s Storm-1 [19] have been developed explicitly to exploit the stream model.

Also, 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 [20] and RAW [21] can be described as polymorphous - that is, they can morph between a number of operating modes, each capturing some class of applications. These polymorphous architectures can be programmed to achieve the stream model's high locality and parallelism.

To program all of these architectures, a number of streaming languages have been developed, including Stream-C/Kernel-C for Imagine, StreamIT [22] for RAW, and the Stream Virtual Machine specification [23], which uses two-level compilation for code written using a supported stream language and a number of architectures (including TRIPS, RAW, Smart Memories, Imagine, and even graphics processing units). There are also performance studies comparing streaming and intelligent memory architectures [24], [25] The stream model has also been more formally studied with respect to other computing models like Kahn process networks and dataflow [26].

IV. Blackfin Processor

The Blackfin is an embedded media processor based on the Micro Signal Architecture [27] developed jointly by Analog Devices (ADI) and Intel. The Blackfin is a fixed-point, convergent architecture that provides both micro-controller (MCU) and digital signal processing (DSP) functionality in a single processor. The MCU functionality is provided with a 32-bit variable-length RISC instruction set supporting vector operations (add/subtract, multiply, shift, etc.) and vector video operations (add/subtract, average, sum-of-absolute-differences, etc.). The DSP functionality is provided via two multiply-and-accumulate (MAC) units accessible via an orthogonal instruction set allowing for up to three instructions to be issued in parallel.

Figure 2 shows the basic units of the Blackfin core: the address arithmetic unit (with appropriate data address and pointer registers, and data address generators), the control unit, and the data arithmetic unit (with data register file, MAC units, vALUs, and ALUs). Also, given the constrained environment in which embedded systems exist, the Blackfin includes software-programmable on-chip PLL, dynamic power management to vary frequency and voltage, multiple operating modes, and memory management unit. The Blackfin hardware supports 8-bit, 16-bit, and 32-bit arithmetic operations - but is optimized for 16-bit operations [28].

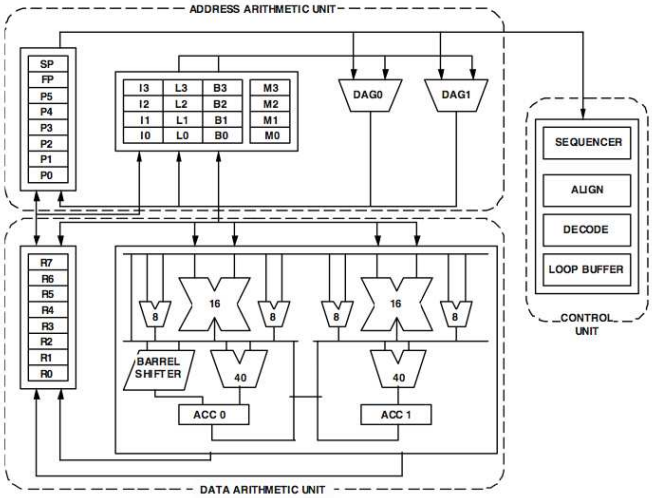

Fig. 2. The Blackfin processor core.

A. Blackfin BF561

The Blackfin derivative used here is the dual-core Blackfin BF561, which is included on the BF561 EZ-KIT LITE evaluation board. The BF561 [29] has the following memory available:

- 64 MB SDRAM main memory in 4 banks of 16MB (4x16MB), operating at up to 133 MHz (system clock)
- 128 KB on-chip L2 (8x16KB), at 300 MHz ($\frac{1}{2}$ core clock)
- 100 KB in-core L1, at 600 MHz (core clock), split into:
  - 32 KB instruction (16 KB SRAM; 16KB configurable as cache or SRAM)
  - 64 KB data (32 KB SRAM; 32 KB cache/SRAM)
  - 4 KB scratch-pad memory

V. Approaches to Using Memory

In our analysis we evaluate three different memory configurations: a worst-case baseline which maps both the original image and the processed image to SDRAM, and two which use low-latency memory. The first low-latency approach copies a partition of the original image to SRAM for each processing kernel, and saves the processed image again to SRAM, which is then written back to SDRAM. The second approach streams the processed subimages between kernels - only reading data from SDRAM when compulsory and writing results to SDRAM when the processing is complete. The following subsections explain these approaches more fully.

A. Per-Kernel Storage

This approach copies subimages into input records, runs a computational kernel, saves results into output records, and saves these records back to SDRAM; this process is then repeated for the next kernel in the control flow of the program. This approach is summarized in the following algorithm:

PK_Transform
1 **foreach** $kernel_i$ **in** Control Flow
2 **do foreach** $SUBIMAGE$ **in** $IMAGE$
3 **do** Copy($SUBIMAGE, kernel_i.in$)
4 Run($kernel_i(kernel_i.in, rows, cols, kernel_i.out)$)
5 Copy($kernel_i.out, SUBIMAGE$)

This approach is similar to performing source transformations of inner-most loops to maximize data locality. Such transformations may cause intra-kernel locality, but inter-kernel communication uses SDRAM, and incurs a per-kernel latency penalty. A side effect of this approach is that the results of each kernel can be saved, but at the cost of increased memory accesses for each kernel.

B. Streaming Local Stores

An alternative approach is to use the stream model. Under this model, SDRAM is only accessed for compulsory reads or completion writes (i.e., only after all n kernels in the control flow have been processed). During the processing of the image, the data *streams* from one kernel to the next, all in low-latency memory. This approach is summarized in the following algorithm:

Stream_Transform
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** Copy($SUBIMAGE, kernel_1.in$)
3 **foreach** $kernel_i$ **in** Control Flow
4 **do** Run($kernel_i(kernel_i.in, rows, cols, kernel_i.out)$)
5 Copy($kernel_n.out, SUBIMAGE$)

Unlike the previous approach, the inter-kernel communication does not use SDRAM and the results of each kernel are not saved - only the final results are saved. For programs where intermediate results are not important, the reduced number of memory accesses provides higher performance.

C. Developing Other Stream Image Processing Programs

Because our approach uses convolution kernel routines, other image processing algorithms (sharpening, embossing, etc.) could be implemented following the same methodology. In order to add more kernels to the stream, we utilized a C data structure representing a kernel, which included pointers to the input/output records and a callback for the kernel's function, and three routines shown below:

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

An initialization using the above routines sets up the control flow of a stream program. Following this initialization, a list traversal of the control flow would call each kernel in the sequence specified. The addition of new kernels or stream mappings only requires a different initialization, without requiring a specialized stream compiler.

VI. Edge Detection Workload

A common problem for automotive computer vision systems is to highlight edges. For example, systems using multiple cameras to perceive road traffic often use software to create a map based on the edge detection of stereo images. This data can be used for lane guidance, vehicle tracking, automated braking control, etc.

Revisiting the generic IMAGEPROCKERNEL algorithm, our program uses 2D convolution as the TRANSFORM subroutine. During convolution, each pixel in the input image is transformed into a weighted sum of its neighbors. The weights are stored in a matrix usually referred to as the "kernel." To avoid confusion with the concept of computational kernels, we will refer to this matrix as a kernel-matrix. The similarity of terms is intentional: kernel-matrices should be stored to low-latency memory because a compute kernel frequently accesses its elements.

Our program performs edge-detection using two convolutions. The first performs a Gaussian blur to reduce noise (such as dust particles in an image), the second performs a Sobel gradient response operation in which the intensity of the resulting image is highest near edges (in either direction). Figure 3 shows the dataflow of this program.

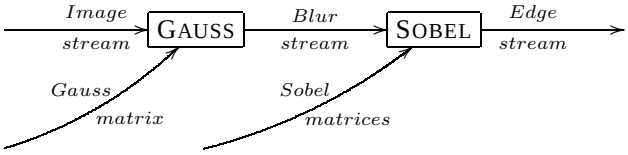

Fig. 3. Dataflow graph for Edge Detection program

To build a stream control flow for the program depicted in Fig. 3, the following initialization calls would be made:

ADDKERNEL(GAUSS)
ADDKERNEL(SOBEL)
NEWSTREAMREC(GAUSS, rows, cols, INPUT)
NEWSTREAMREC(GAUSS, rows, cols, OUTPUT)
NEWSTREAMREC(SOBEL, rows, cols, OUTPUT)
MAPSTREAM(GAUSS, OUPUT, SOBEL, INPUT)

VII. Implementation and Results

We implemented an edge detector in C using the Visual DSP++ 4.0 integrated development and debugging environment, and the web tutorial provided by [30] to convolve bitmap (BMP) source images with the kernel-matrices below: $\mathbf{Gauss}$ for blurring [31], and $\mathbf{Sobel_x}$, $\mathbf{Sobel_y}$ for vertical and horizontal edge detection [32].

$$\mathbf{Gauss} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\mathbf{Sobel_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{Sobel_y} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

To make use of the Blackfin's dual-MAC DSP instruction set, we used and adapted the Sobel assembly provided in the Blackfin SDK 2.0 [33]. For the Gaussian blur kernel, two pixels were processed concurrently; for the Sobel kernel, both horizontal and vertical convolutions were performed in parallel.

For edge detection, image boundaries are important. Using a partitioned image introduces erroneous edges due to the borders of the resulting sub-images. Both convolutions used 3x3 kernel-matrices, so each partitioned subimage needed to access the last row of the previous subimage. A moving window of "live" data was used; data was copied into SRAM, processed, saved, and the window re-addressed to repeat the process for the next sub-image.

Our input dataset is a 1920x1080 pixel (WxH) image extracted from a HD video. Below we describe the memory configuration used and the results obtained (presented in Tables I-IV. All results are generated by running on the real (versus simulated) Blackfin hardware and are the average of five trials per configuration.

A. Baseline Approach

The baseline represents the worst-case because the memory accesses are all mapped to SDRAM, as depicted in the following algorithm:

EDGEDETECT
1 RUN(GAUSS($origImage, rows, cols, blurImage$))
2 RUN(SOBEL($blurImage, rows, cols, edgeImage$))

$origImage$, $blurImage$, and $edgeImage$ are pointers to main memory. Table I shows the baseline results: despite the use of both MAC units and hand-coded assembly, the total runtime is reduced from 3.41 to 2.48 sec - only a 27% reduction. This result demonstrates that utilizing the memory hierarchy effectively is, for our image processing workload, more important than effectively using the computational resources.

TABLE I
BASELINE: WORST-CASE MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	0 (0)	0 (0)
Gauss convolution	954, 852, 092 (1.59)	568, 957, 265 (0.95)
Sobel convolution	1, 092, 902, 630 (1.82)	921, 313, 359 (1.54)

B. Per-Kernel Approach

Expanding on the algorithm PK_TRANSFORM, the per-kernel approach utilizes the BF561 by mapping each kernel's input and output records to L2 or L1 SRAM.

PK_ EDGEDETECT
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** COPY($SUBIMAGE$, GAUSS.in)
3 RUN(GAUSS(GAUSS.in, $rows$, $cols$, GAUSS.out))
4 COPY(GAUSS.out, $SUBIMAGE$)
5 **foreach** $SUBIMAGE$ **in** $IMAGE$
6 **do** COPY($SUBIMAGE$, SOBEL.in)
7 RUN(SOBEL(SOBEL.in, $rows$, $cols$, SOBEL.out))
8 COPY(SOBEL.out, $SUBIMAGE$)

In Table II, the effect of using L2 is shown. This approach introduces a delay due to copying image data into and out of L2, but shows that localizing data accesses impacts total runtime that is less than the Sobel convolution alone, in the worst-case. Also, the effect of using both MAC units is more pronounced, the total runtime is decreased from 1.77 to 0.84 sec - a reduction of 53%, almost double the baseline reduction.

TABLE II
PER-KERNEL L2 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	257, 576, 841 (0.43)	127, 812, 414 (0.21)
Gauss convolution	353, 170, 263 (0.59)	174, 660, 214 (0.29)
Sobel convolution	450, 098, 388 (0.75)	198, 909, 374 (0.33)

Table III shows that even though the copy time is about the same ($\sim$0.20 sec for the assembly implementation) due to the SDRAM latency, the convolution runtimes can be reduced with L1 accesses at core-clock speed.

TABLE III
PER-KERNEL L1 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	233, 324, 154 (0.39)	112, 447, 043 (0.19)
Gauss convolution	198, 205, 503 (0.33)	26, 998, 728 (0.04)
Sobel convolution	300, 954, 794 (0.50)	48, 514, 428 (0.08)

C. Stream Approach

Expanding on the algorithm STREAM_TRANSFORM, the stream and per-kernel approaches both map each kernel's input and output records to L2 or L1 SRAM. But the stream approach only accesses SDRAM when compulsory or when data has been completely processed and is ready to be saved.

STREAM_ EDGEDETECT
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** COPY($SUBIMAGE$, GAUSS.in)
3 RUN(GAUSS(GAUSS.in, $cols$, GAUSS.out))
4 RUN(SOBEL(GAUSS.out, $rows$, $cols$, SOBEL.out))
5 COPY(SOBEL, $SUBIMAGE$)

In Table IV the effects of using the stream model are shown. This approach reduces the copy delay because only compulsory or complete accesses occur. The convolution runtimes remain about the same compared to the per-kernel approach, but the overall runtime is reduced.

TABLE IV
STREAM L2 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	128, 774, 780 (0.21)	63, 907, 125 (0.11)
Gauss convolution	349, 059, 870 (0.58)	174, 687, 337 (0.29)
Sobel convolution	450, 126, 228 (0.75)	198, 936, 082 (0.33)

Table V shows that again the stream model provides the best runtime, with the convolution runtimes being about the same as in the per-kernel approach. In the next subsection we compare the approaches and summarize the results.

TABLE V
STREAM L1 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	116, 647, 351 (0.19)	56, 225, 653 (0.09)
Gauss convolution	194, 094, 567 (0.32)	27, 028, 002 (0.05)
Sobel convolution	300, 982, 290 (0.50)	48, 540, 970 (0.08)

D. Summary

Tables VI and VII show the combined results for the three approaches, ordered by decreasing execution time in seconds, for both the C and assembly implementations.

TABLE VI
COMPARING MEMORY UTILIZATION APPROACHES WITH C IMPLEMENTATION.

	Total Runtime, sec	Speedup compared to Baseline
Baseline	3.41	1.00
Per-Kernel L2	1.77	1.93
Stream L2	1.55	2.20
Per-Kernel L1	1.22	2.80
Stream L1	1.02	3.34

TABLE VII
COMPARING MEMORY UTILIZATION APPROACHES WITH ASSEMBLY IMPLEMENTATION.

	Total Runtime, sec	Speedup compared to Baseline
Baseline	2.48	1.00
Per-Kernel L2	0.84	2.95
Stream L2	0.73	3.40
Per-Kernel L1	0.31	8.00
Stream L1	0.22	11.3

Accessing both MAC units shows little performance benefit for the baseline approach using only SDRAM. The total runtime is reduced from 3.41 to 2.48 sec, a mere 27% decrease attained by using hand-coded assembly. The penalty for using

higher latency memory outweighs the benefits of using the computational resources efficiently.

However when L2 is used, the assembly's power is shown as the edge detector runs over 2X faster for both low-latency approaches (and either implementation). Using L1 is even more beneficial, providing over 3X faster runtimes. Even higher performance is achieved by using the stream approach which uses low-latency meory as often as possible.

VIII. Related Work

Previous work used the Blackfin's configurable memory focused on mapping code to SRAM and a code layout tool [34]. Recently, frameworks to develop optimized media applications on the Blackfin have described a set of "templates" which exploit predictable data access patterns to make use of low-latency memory. [35], [36], [37]

The stream processing paradigm [38] is naturally suited to media applications [39], [40], which display large amounts of parallelism, little reuse of data, and a high ratio of computations to memory accesses. Some scientific applications have similar characteristics but a difference lies in memory access patterns. Whereas media applications fit well into streams because the gathering of data will be more or less sequential, scientific applications often need more random access to memory.

Despite this difference, Gummaraju and Rosenblum [41] found a 27% performance increase using the stream paradigm for certain scientific applications. The architecture used was the Intel Pentium 4 and the performance was limited due to the cache overhead of non-temporal loads and stores. The Blackfin has a configurable memory hierarchy and stands to benefit by using low-latency memory, without the overhead of maintaining the cache.

The Data Transfer and Storage Exploration methodology [11] performs data-dependence analysis and performs source-to-source code transformations (such as loop transformations) to ensure optimizations for the memory hierarhcy (for example, reorganizing inner and outer loops to improve producer-consumer locality). The stream model incorporates some of the same lessons of the DTSE project and its programming style might serve to reduce some of the loop transformations.

IX. Conclusion

In this paper, we examined the use of the configurable memory hierarchy on the Blackfin BF561, the use of the two MAC units for image processing by matrix convolution, and the application of the stream model to use low-latency SRAM.

We found that for both L2 and L1 SRAM, using the stream model in which accesses to SDRAM are limited to compulsory and completed reads and writes, respectively, can achieve speedups of 2.8X to 3.3X versus a worst-case baseline that uses SDRAM exclusively using C; and speedups of 3.4X to 11.3X using hand-optimized assembly code available from ADI.

The stream model discussed above provides a natural means to express image processing programs such that the memory hierarchy is effectively used (i.e., on-chip memory is used for the majority of data streams). The model is simple enough to be implemented using basic C data structures and callbacks to represent control flow, but powerful enough to achieve an order of magnitude speedup compared to the worst-case baseline approach.

X. Future Work

The Stream Virtual Machine [23] described above contains master control and slave kernel processors, and DMA units. The Blackfin supports dual cores and a DMA engine. If a low-level compiler for the SVM were developed to support it, then programs written in stream languages could potentially perform better on the Blackfin. Other embedded systems supporting similar features could also stand to benefit from the use of the stream model.

Acknowledgment

The authors would like to thank Mimi Pichey, Giuseppe Olivadoti, Richard Gentile, and Ken Butler from Analog Devices for their generous donation of Blackfin EZ-KIT Lite evaluation boards, software, and extender boards, and for their support of this project. We also extend our gratitude to the anonymous reviewers for their comments and suggestions.

References

[1] Wm. A. Wulf and Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. SIGARCH Computer Architecture News, 23(1):2024, 1995.

[2] Sally A. McKee. *Reflections on the Memory Wall*. In Proceedings of the 1st Conference on Computing Frontiers (CF '04), page 162, New York, NY, USA, 2004.

[3] Rita Cucchiara, Massimo Piccardi, and Andrea Pati. *Exploiting Cache in Multimedia* In Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS '99), Florence, Italy, 1999.

[4] Andrea Pati. *Exploring Multimedia Applications Locality to Improve Cache Performance* In Proceedings of 8th ACM International Conference on Multimedia (MULTIMEDIA '00), pp. 509-510. Marina del Rey, California, USA, 2000.

[5] Afrin Naz, Krishna Kavi, Philip Sweany, and Mehran Rezaei. *A Study of Separate Array and Scalar Caches*, In Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS'04), pp. 157-164, Winnipeg, Manitoba, Canada, 2004.

[6] Afrin Naz, Mehran Rezaei, Krishna Kavi, and Philip Sweany. *Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers*, In Proceedings of the 2004 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA-2004), Antibes Juan-les-Pins, France, 2004.

[7] Nimrod Megiddo, and Dharmendra Modha. *Outperforming LRU with Adaptive Replacement Cache Algorithm*, Computer, 37(4):58-65, 2004.

[8] Nimrod Megiddo, and Dharmendra Modha. *ARC: A Self-Tuning, Low Overhead Replacement Cache*, In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03), pp. 115-130, San Francisco, CA, USA, 2003.

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. *Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems* In Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES 2002), pp.73-78, Estes Park, Colorado, USA, 2002.

[10] Monica Lam and Michael Wolf. *A Data Locality Optimizing Algorithm* In Proceedings of the 12th Conference on Programming Language Design and Implementation (PLDI 1991), pp. 30-44, Toronto, Ontario, CA, 1991.

[11] Francky Cathoor, Koen Danckaert, Sven Wuytack, and Nikil D. Dutt. *Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors* IEEE Design&Test of Computers 18(3):70-82, 2001.

[12] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick . *A Case for Intelligent RAM* IEEEmicro 17(2):34-44, 1997.

[13] Sunaga, T, Peter M. Kogge, et al, *A Processor In Memory Chip for Massively Parallel Embedded Applications*, IEEE Journal of Solid State Circuits, Oct. 1996, pp. 1556-1559.

[14] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. *Stream Processors: Programmability and Efficiency* ACM Queue, 2(1):52-62, 2004.

[15] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. *Programmable Stream Processors*, ACM Computer 26(8):54-62, 20003.

[16] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. *Imagine: Media Processing with Streams*. IEEE Micro, 21(2):35-46, 2001.

[17] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. *Evaluating the Imagine Stream Architecture*. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA 2004), pp. 14-26, Munchen, Germany, 2004.

[18] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. *Merrimac: Supercomputing with Streams*, Proceedings of the ACM/IEEE conference on Supercomputing (SC '03) p. 35, Phoenix, Arizona, USA, 2003.

[19] Stream Processors, Inc., 455 DeGuigne Drive Sunnyvale, CA 94085, USA. *Stream Processing: Enabling a new class of easy to use, high-performance parallel DSPs*, revision 1.9, 2007, Document: SPI_MWP.

[20] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. *TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP* ACM Trans. on Architecture and Code Optimization, 1(1):62-93, 2004.

[21] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Sama Amarasinghe, and Anant Agarwal. *Bringing It All to Software: Raw Machines*, IEEE Computer, 30(9):86-93, 1997.

[22] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. *A Stream Compiler for Communication-Exposed Architectures*. SIGPLAN Not. 37(10):291-303, 2002.

[23] *The Stream Virtual Machine*. Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'2004), pp. 267-277, Antibes Juan-les-Pins, France, 2004.

[24] Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. *A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels*, In Proceedings of the 30th International Symposium on Computer Architecture (ISCA 2003), pp. 410-421, San Diego, CA, USA, 2003.

[25] Sourav Chatterji, Manikandan Narayanan, Jason Duell, and Leonid Oliker. *Performance evaluation of two emerging media processors: VIRAM and Imagine* In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 2003), pp. 229.1, Nice, France, 2003.

[26] Bart Kienhuis and Ed F. Deprettere. *Modeling Stream-Based Applications Using the SBF Model of Computation*, Journal of VLSI Signal Processing Systems, July 2003, pp. 291-300.

[27] *Micro Signal Architecture from ADI and Intel*. `www.intel.com/design/msa/highlights.pdf`,

[28] Analog Devices, Inc., One Technology Way, Norwood, MA 02062, USA. *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*, revision 1.0 edition, June 2005, Part Number 82-000556-01.

[29] Analog Devices, Inc., One Technology Way, Norwood, MA 02062. *ADSP-BF561 Blackfin Processor Hardware Reference*, revision 1.0 edition, July 2005, Part Number 82-000561-01.

[30] Bill Green. *Edge Detection Tutorial*. `www.pages.drexel.edu/~weg22/edge.html`, 200 2.

[31] *Efficient algorithm for Gaussian blur using finite-state machines* Frederick M. Waltz and John W. V. Miller, Proc. SPIE Int. Soc. Opt. Eng. 3521, 3 34-341, 1998.

[32] Bob Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. *Sobel Edge Detector* as part of Univ. of Edinburgh HyperMedia Image Proces sing Reference, `www.cee.hw.ac.uk/hipr/html/sobel.html`, 1994.

[33] Analog Devices, Inc. *Software Development Kit (SDK) Downloads*, `www.analog.com/processors/platforms/sdk.html`, SDK Rel 2.00, January 2007.

[34] Kaushal Shanghai, David Kaeli, and Richard Gentile. *Code and Data Partitioning on the Blackfin 561 Dual-Core Platform*. In Proceedings of the 3rd Workshop on Optimizations for DSP and Embedded Systems, pp. 92-100, San Jose, CA, 2005.

[35] Kunal Singh and Ramesh Babu. *Video Framework Considerations for Image Processing on Blackfin Processors*. Analog Devices Inc., App Note EE-276, Rev 1, Sept 2005.

[36] Kaushal Sanghai. *Video Templates for Developing Multimedia Applications on Blackfin Processors*. Analog Devices Inc., App Note EE-301, Rev 1, Sept 2006.

[37] Glen Ouellette, and Kaushal Sanghai. *Efficient Data Management Frameworks for Multimedia Applications*, DSP-FPGA.com, Dec 2006. `Www.dsp-fpga.com/articles/ouellette_and_sanghai/`

[38] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. *A Bandwidth-Efficient Architecture for Media Processing*. In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31), pp. 3-13, Los Alamitos, CA, USA, 1998.

[39] Ruby B. Lee and Michael D. Smith. *Guest Editor's Introduction: Media Processing: A New Design Target*. IEEE Micro, 16(4):6-9, 1996.

[40] Keith Diefendorff and Pradeep K. Dubey. *How Multimedia Workloads Will Change Processor Design*. Computer, 30(9):43-45, 1997.

[41] Jayanth Gummara ju and Mendel Rosenblum. *Stream Programming on General-Purpose Processors*. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38), pp. 343-354, Washington, DC, USA, 2005.