

Multicore-programming frameworks for embedded-multimedia applications

UNDERSTANDING THE DATA-ACCESS PATTERN OF AN APPLICATION CAN HELP YOU EFFECTIVELY USE THE MEMORY AND SYSTEM RESOURCES OF THE UNDERLYING ARCHITECTURE TO DEVELOP A SCALABLE PARALLEL APPLICATION.

It is becoming difficult to meet the growing processing demands of embedded-multimedia applications with single-core architectures. Although multicore embedded architectures have emerged as a promising solution to this problem, they bring with them the challenge of developing software that efficiently uses them. Current compiler technology and development tools require more sophistication to make multicore architectures successful. You develop most parallel software by converting sequential programs to parallel programs by hand, and a lack of multicore-aware developmental tools makes the software difficult to evaluate. Without solid project planning up-front, you may be facing inefficient applications and an increased time to market.

Software frameworks can provide a better starting point for developing multicore applications and thus help to reduce the development time. This article demonstrates frameworks for embedded-multimedia applications; however, you can extend the data-flow models to many other applications. The frameworks incorporate the inherent data parallelism in multimedia applications and demonstrate effective management of streaming data by efficiently using the underlying architecture.

There are two significant challenges to producing parallel software in which you can scale the performance of a sequential application to the number of available cores: developing efficient parallel algorithms and efficiently using the shared resources such as the memory, DMA (direct-memory-access) channels, and interconnect network.

There are often several ways to parallelize an application. Some applications exhibit inherent parallelism; others have extremely complex and irregular data-access patterns. In general, scientific applications and multimedia applications are often easier to parallelize, because their data-access patterns are more predictable than those of control applications. This article discusses parallelization techniques targeting multimedia algorithms, which require high processing power and are attractive for embedded-system applications.

Levels of data parallelism exist in multimedia applications. The granularity of parallelism varies greatly from a set of frames to a macroblock of a frame. In general, the lower the granularity, the higher the level of synchronization you need between the sharing elements—cores and DMA channels, for example. Lower granularities increase parallelism and reduce

network traffic; higher granularities require lower synchronization but also increase the network traffic. So, based on the application type and system requirements, the frameworks define different levels of parallelism.

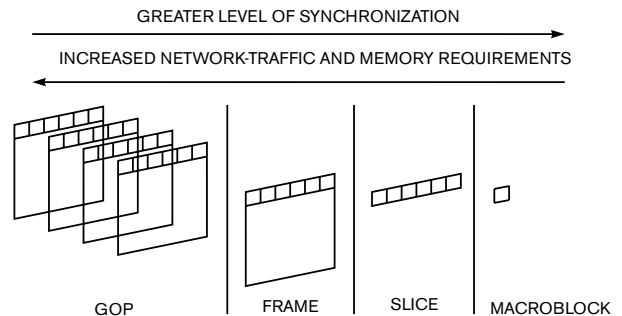


Figure 1 Multimedia applications exhibit various levels of data parallelism that result in corresponding trade-offs.

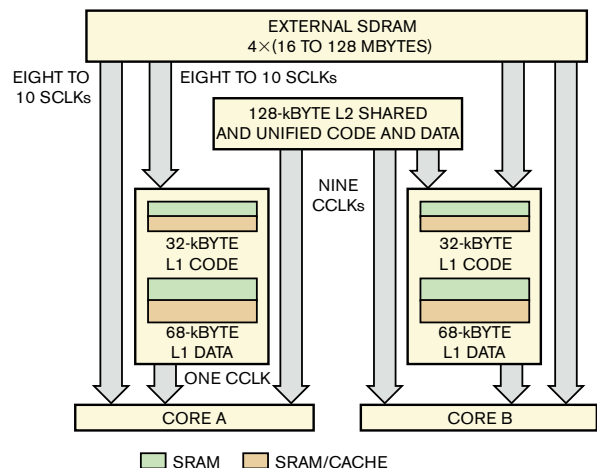


Figure 2 The ADSP-BF561 architecture consists of separate instruction and data memory private to the two cores and a shared L2 and external memory.

As noted, developing scalable parallel software also depends on the efficient use of the interconnect network, memory hierarchy, and the peripheral/DMA resources. The strict low-power and low-cost requirements of a system constrain all of these elements. Efficient use of these resources when programming in a multicore environment requires innovation. This article presents some ideas to help efficiently manage these resources on Analog Devices' Blackfin ADSP-BF561 dual-core processor.

MULTIMEDIA-DATA-FLOW ANALYSIS

To achieve data parallelism, the goal is to find a block of data or a set of blocks of data in the streaming data that you can treat independently and "feed" to a processing element. Independent blocks of data reduce synchronization overhead and make parallelizing algorithms easier. To find this data, it is important to understand the data-flow model, or the "data-access pattern," of an application.

For most multimedia applications, you can view the data-access pattern as a 2-D pattern (spatial domain), in which the independent blocks of data are confined to a single frame, and a 3-D pattern (temporal domain), in which the independent blocks of data span more than one frame. In the spatial domain, you can divide the frame into slices with N sequential rows and macroblocks of a video frame. In the temporal domain, you can subdivide the data flow at a frame level or a GOP (group-of-pictures) level.

Algorithms with a slice or macroblock data-access pattern require greater synchronization but have less network traffic, as the memory hierarchy needs to store only a part of the image data. In the case of a frame- or a GOP-type data-access pattern, the memory hierarchy needs to store large amounts of data but requires considerably less synchronization, as the system exhibits higher granularities of parallelism. **Figure 1** shows levels of parallelism that exist in a multimedia application; it correspondingly shows the relative synchronization and the network traffic between the four levels.

MULTICORE-ARCHITECTURE ANALYSIS

Figure 2 shows the ADSP-BF561 architecture, which consists of separate instruction and data memory private to the two cores and a shared L2 and external memory. You can interface all peripherals and DMA resources to either core with configurable-arbitration schemes. There are two DMA controllers, each of which consists of two sets of MDMA (memory-DMA) channels. A separate bus connects the L2 memory and each core. A shared bus connects the external memory and the two cores.

All of the frameworks use DMA to move the streaming data within the memory hierarchy. The other alternative, cache memory, does not manage any data. You know the data-access pattern for the applications you are targeting; thus, you can effectively use the DMA engine to manage data. The cache suffers from nondeterministic access times, cache-miss penalties, and increased external-memory-bandwidth requirements. Using the DMA engine, you can transfer data to L1 memory before a core request; the system performs transfers in the back-

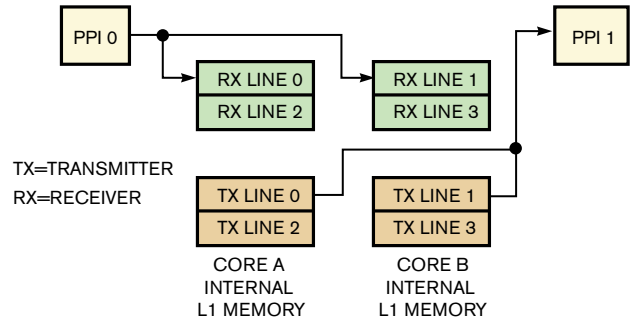


Figure 3 In the line-processing framework, Core A handles the video input, and Core B manages video output.

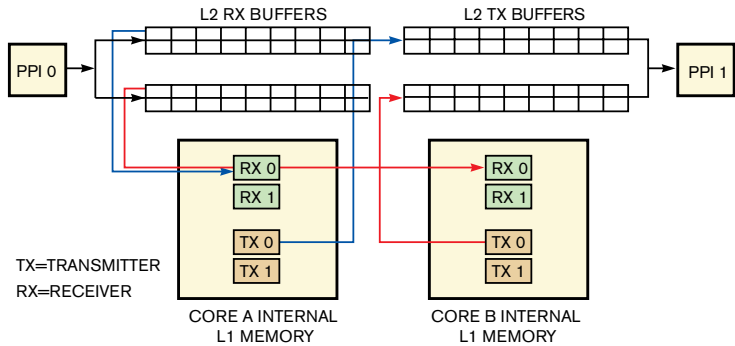


Figure 4 In the dual-core macroblock data-flow model, the L2 memory maintains multiple slice buffers, and separate MDMA channels transfer macroblocks from L2 to L1 memory of each core.

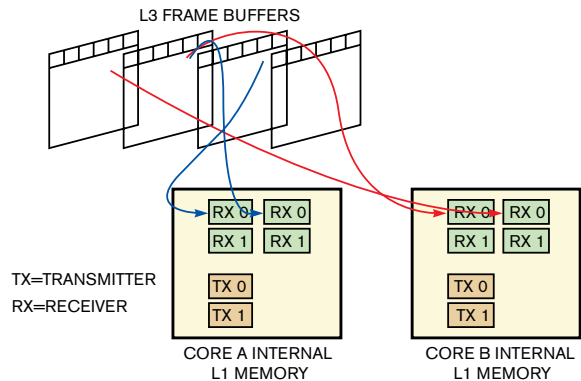


Figure 5 In frame-level processing, external memory stores the dependent frames.

ground without halting the core for a data-item request.

With two sets of MDMA channels on each DMA controller, the system evenly distributes MDMA channels among the cores for symmetrical parallel processing.

You can easily exploit the fast access time of L1 and L2 memories for applications with smaller granularities of data-access patterns. You can directly transfer the independent blocks of data to L1 or L2 memory from the peripheral interface without accessing the slow external memory—saving valuable external-memory bandwidth and MDMA resources and reducing data-transfer time.

For applications in which higher levels of granularity char-

acterize the data-access pattern, memory becomes a bottleneck, as the smaller L1 and L2 memory levels are insufficient to hold multiple frames of data. However, although the data dependency exists between multiple frames, the dependency is more often than not over only the smaller blocks across frames. If you could store all the dependent frames in a larger memory space (external memory), then you could sequentially transfer only the independent blocks from each frame to the available cores for processing. If these independent blocks of data are considerably smaller than the frame data, so as to fit in the memory space of L1 or L2, you can efficiently process the data with lower memory-access latency.

Two cores share the L2 and external-memory-interface bus, although separate buses connect both memory levels. Thus, you should minimize simultaneous access by the two cores to the same memory level to avoid stalls due to contention. To minimize contention, the frameworks map code and data objects such that only one core maximally accesses the L2 core; the other core maximally accesses external memory. In this case, the core performing most of the external-memory accesses has greater memory-access latency, but overall access latency is less than the cost of contention.

The framework assigns all input-peripheral interfaces to one core and all output-peripheral interfaces to the other core. The frameworks use a video-in/out example using the PPI (parallel peripheral interface) to input and output video frames. The BF561 architecture has two PPIs.

If the interrupt processing time is less than the processing time of the streaming data, you can assign all peripheral interfaces to one core for ease of programming; the lower interrupt processing time will not affect the load balancing between the two cores.

PROPOSED FRAMEWORK MODEL

Based on the granularity of the data-access pattern, you can define four frameworks: line processing (spatial), macroblock processing (spatial), frame processing (temporal), and GOP processing (temporal). If the data-access pattern of an application fits one of these four models, you can use the corresponding framework. There are also ways to integrate multi-

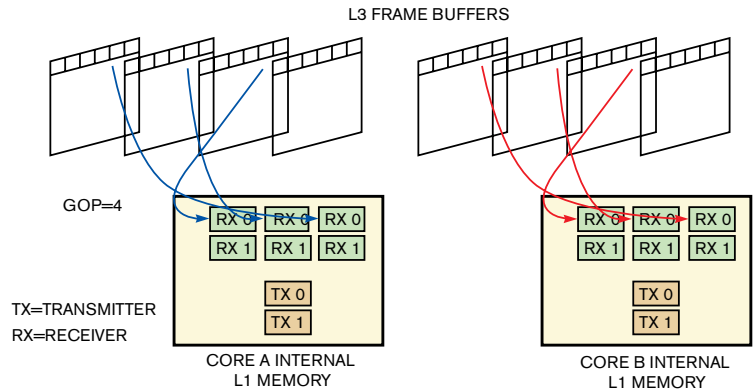


Figure 6 For a GOP-type data-access pattern, dependency exists within a set of frames, and there is no data dependency between two sets of frames.

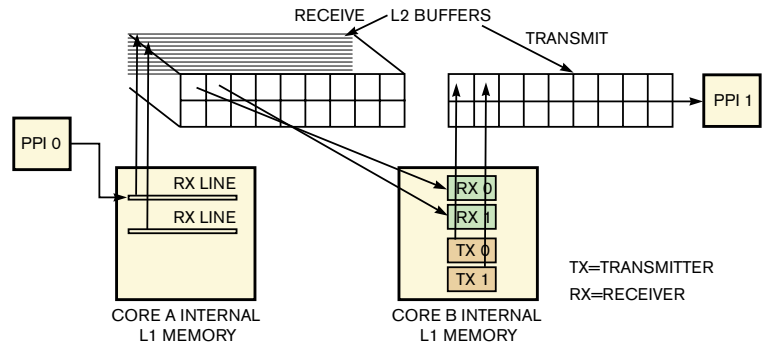


Figure 7 The data flow combines the line- and macroblock-processing frameworks.

ple frameworks for asymmetrical parallel processing when you have two or more processing algorithms for a data stream.

In the case of line processing, dependency exists only at a line level—that is, between adjacent pixels. Every line forms a data block that each core can independently process. Figure 3 shows the data-flow model for the line-processing framework. Core A handles the video input, and Core B manages video output. Separate sets of MDMA channels manage data between Core A and Core B. L1 memory uses multiple buffers to avoid contention between core and peripheral-DMA access. A counting semaphore ensures that every line achieves synchronization between the two cores. A single-core imple-

TABLE 1 FRAMEWORK SPECIFICATIONS

Template	Core cycles/ pixel× approximately single core	Core cycles/ pixel× approximately two cores	L1 data memory required (bytes)	L2 data memory required (bytes)	Comments
Line processing	42	80	Line size×2; 1716×2 for ITU-656		Double buffering in L1
Macroblock processing	36	72	Macroblock size N×M×2	Slice of a frame; (macroblock height×line size)×4	Double buffering in L1 and L2
Interframe processing	35	70	Size of subprocessing block×number of dependent blocks	Size of subprocessing block×number of dependent blocks	Only L1 or L2 can be used, double buffering in L1 or L2

mentation of this framework also takes advantage of moving data directly into the L1 memory, thereby saving external-memory bandwidth and DMA resources. Examples of applications that can use this framework include color conversion, histogram equalization, filtering, and sampling.

Figure 4 shows the data-flow model for the macroblock-data-access pat-

tern. You can move alternate macroblocks between the two cores. The L2 memory maintains multiple slice buffers, and separate MDMA channels transfer macroblocks from L2 to L1 memory of each core. L1 memory also maintains multiple buffers to avoid contention between DMA and core access. Similar to the line-processing framework, Core A handles the input-video interface, and

Core B manages the output interface; a counting semaphore achieves synchronization between the two cores. Targeted example applications for this framework include edge detection, JPEG/MPEG-encoding/decoding algorithms, and convolution encoding.

In frame-level processing, external memory stores the dependent frames. Depending on the granularity of dependency between frames (macroblock or line), the system transfers subblocks of frames to the L1 or L2 memory. **Figure 5** shows the data flow for the frame-level-processing framework. In this case, assuming a macroblock dependency across multiple frames, the system transfers macroblocks of frames to the L1 memory. Similar to the other frameworks, Core A handles the input-video interface, and Core B manages the output interface. A counting semaphore achieves synchronization between the two cores. Example applications for this framework include motion-detection algorithms.

In GOP-level processing, each core processes multiple sequential frames. The difference between the frame-processing framework and the GOP-level framework is that the frame-processing one performs spatial division within frames, whereas the GOP-level one uses temporal division (sequence of frames) to implement parallelism. For a GOP-data-access pattern, dependency exists within a set of frames, and there is no data dependency between two sets of frames. Thus, cores can independently process each set. **Figure 6** shows the data flow for this framework. Similar to the frame-processing framework, the system can transfer blocks of frames to the L1 memory of the cores. To efficiently use the interleaved memory-banking structure of the external memory, the system equally divides the banks among the cores. Each external bank of the ADSP-BF561 supports as many as four internal-SDRAM banks. Examples of applications that could use this framework include encoding/decoding algorithms, such as MPEG-2/4.

In a more real-world application, multiple algorithms running within the system process the streaming data, and each of these algorithms may exhibit a different data-access pattern. In such cases, you can combine the frameworks for a particular application. To take advantage of the multiple cores you can pipe-

line the process to achieve parallelism. This type of parallelism is asymmetrical because there could be unequal computation on the cores. However, the system can allocate several other tasks to unused instructions of a core to achieve load balancing and still maintain flexibility. **Figure 7** shows the data-flow model for a combination of line- and macroblock-processing frameworks.

In several other applications, data dependency would exist across multiple blocks of data. The data-access pattern is still predictable, but it extends beyond the granularities of a macroblock or a line. For example, a motion-window search uses several adjacent macroblocks. The data-access pattern is still predictable, but

MORE AT EDN.COM ▶

+ Go to www.edn.com/ms4275 and click on Feedback Loop to post a comment on this article.

the system accesses blocks of data between several iterations of an algorithm. In such cases, you can modify the frameworks to achieve efficient parallelism. For example, if dependencies exist between several lines, you can modify the line-processing framework to transfer slices of frames of N sequential lines to the L1 memory of each core. In a similar way, you can extend the macroblock-processing framework to transfer more than one macroblock to the internal L1 memory from the L2 memory.

FRAMEWORK ANALYSIS

To evaluate the dual-core frameworks, Analog Devices first developed a single-core application with the data-flow model and then compared it with the dual-core implementation. **Reference 1** discusses the single-core models in more detail. Blackfin-specific system-optimization techniques can also efficiently use the available bandwidth (**Reference 2**). To keep the analysis simple, the company compared only the speed of the basic frameworks and not the combination of frameworks.

The cycles are the core computation cycles available for processing the stream data to meet real-time constraints for an NTSC (National Television Systems Committee) video input. For a core running at 600 MHz, the total cycles available per pixel to meet the real-time constraints is 44 cycles/pixel. Any core access to the stream data is only a single-core cycle, as all data access is to L1 memory. The cycles shown also exclude any interrupt latency.

As **Table 1** shows, the dual-core frameworks effectively double the speed on all the frameworks. The **table** also shows the L1-memory usage for each core and the shared-memory space that each framework requires. The frameworks use the Analog Devices DD/SSL (device driver/system services library) for peripheral and data management (**Reference 3**). **EDN**

REFERENCES

- 1 "System Optimization Techniques for Blackfin Processors (EE-324)," Revision 1, July 2007, Analog Devices Inc.
- 2 "Video Templates for Developing Multimedia Applications on Blackfin Processors (EE-301)," Revision 1, Septem-

ber 2006, Analog Devices Inc.

3 "Device Drivers and System Services Manual for Blackfin Processors," Revision 2.0, March 2006, Analog Devices Inc.

4 "ADSP-BF533 Blackfin Processor Hardware Reference," Revision 3.1, May 2005, Analog Devices Inc.

5 "ADSP-BF561 Blackfin Processor Hardware Reference," Revision 3.1, May 2005, Analog Devices Inc.

6 Katz, David and Rick Gentile, *Embedded Media Processing*, Newnes Publishers, Burlington, MA, 2005.

7 Poynton, Charles, *Digital Video and HDTV*, Morgan Kaufmann Publishers Inc, San Francisco, 2003.

8 "Video Framework Considerations for Image Processing on Blackfin Processors," Revision 1, September 2005. Analog Devices Inc.

AUTHORS' BIOGRAPHIES

Kaushal Sanghai has been a Blackfin applications engineer at Analog Devices since 2006. He holds a master's degree in electrical engineering from Northeastern University (Boston) and a bachelor's degree in electronics engineering from Mumbai University, India. His areas of interest include embedded-system design, computer architecture, and parallel computing.

Rick Gentile joined Analog Devices in 2000 as a senior DSP-applications engineer, and he currently leads the processor-applications group. He is co-author of Embedded Media Processing. Before joining Analog, Gentile was a member of the technical staff at the Massachusetts Institute of Technology Lincoln Laboratory, where he designed several signal processors for a range of radar sensors. He has a bachelor's degree from the University of Massachusetts—Amherst and a master's degree in electrical and computer engineering from Northeastern University (Boston).

David Katz has more than 15 years of experience in analog, digital, and embedded-system design. Currently, he is Blackfin applications manager at Analog Devices, where he is involved in specifying and supporting processor-based embedded designs. He is co-author of Embedded Media Processing. Previously, he worked at Motorola as a senior design engineer in cable-modem and automation groups. Katz holds both bachelor's and master's degrees in electrical engineering from Cornell University (Ithaca, NY).