Multi-core programming frameworks for embedded multimedia applications

Kaushal Sanghai and Rick Gentile
Analog Devices Inc.
Norwood, MA
{kaushal.sanghai, richard.gentile}@analog.com

Introduction

The growing processing demands placed by embedded multimedia applications are getting increasingly difficult to meet with single-core architectures. In an effort to increase the performance for single core architectures, the power dissipation is reaching unsustainable levels as we try to pack more and more transistors on a single chip and increase the clock speed further. To overcome this barrier, multi-core embedded architectures have shown the promise to balance the high performance and low power requirements for embedded applications. But there hasn't been much evidence of success in utilizing them efficiently because of the limited availability of parallel software, compiler technology and development tools for multi-core architectures. This is forcing application developers to convert sequential programs to parallel software by hand in addition to the fact that it is harder to evaluate a parallel application due to the lack of developmental tools. This makes developing parallel applications extremely difficult and time consuming thereby increasing the time to market multi-core embedded systems.

Considering the above, frameworks can provide a better starting point for developing multi-core applications and thus help to reduce the time of development. In this paper, we demonstrate frameworks for embedded multimedia applications although the data flow models applied within the frameworks can be extended to many other applications. The frameworks described incorporate the inherent data parallelism in multimedia applications and also demonstrate effective management of streaming data by efficiently utilizing the underlying architecture.

The frameworks exploit the natural data parallelism exhibited by multimedia applications. Levels of data parallelism exist in multimedia applications. The granularity of parallelism varies greatly from a set of frames to a macro-block of a frame. Lower the granularity, higher the synchronization required between the sharing elements (cores, DMA channels etc.) although lower granularities give increased parallelism and reduced network traffic.

Developing scalable parallel software also greatly depends on the efficient use of the interconnect network, memory hierarchy, and the peripheral/DMA resources. All the above elements are constrained by the strict low power and low cost requirements of a system. Thereby innovative ways are needed to efficiently utilize these resources when programming in a multi-core environment. We present some novel ideas on managing these resources efficiently on the Blackfin 561 dual-core platform.

To summarize, our framework focuses on the following key aspects of multi-core programming:

1. Parallelizing programs:
   - Understanding inherent parallelism within applications and its data flow analysis.
2. Managing shared resources:
   - Managing shared resources such as memory and peripheral/DMA channels
3. Synchronization:
   - Managing shared code and data memory,
   - Managing multiple heaps and stacks

The following few sections describe the ADSP-BF561 dual-core architecture and the frameworks.

ADSP-BF561 architecture

We briefly describe the ADSP-BF561 architecture. It consists of separate code and data memory, private to the two cores and a shared L2 and external memory. All peripherals and DMA resources can be interfaced to either core with configurable arbitration schemes. There are two DMA controllers; each of which consists of 2 Memory DMA channels. Internal memory DMA channel is also available to transfer data between L1 and L2 memory at a much faster rate. The bus connecting the L2 memory and the two cores is shared. Another bus connects the external memory and the two cores but is also shared. There are no crossbars or switches on any interconnect bus.

Frameworks

To achieve data parallelism, the goal is to find a block of data or a set of blocks of data in the stream data that can be treated independently for feeding to a processing element. Finding independent blocks of data reduces synchronization overhead and makes parallelizing algorithms easier. To find these independent blocks of data it is important to understand the data flow model or the *data access pattern* of an application.

For most multimedia applications, the data access pattern can be viewed as a 2-d pattern (spatial domain) where the independent blocks of data are confined to a single frame and 3-d pattern (temporal domain) where the independent blocks of data span more than one frame. In the spatial

domain, the frame can be divided into slices (n sequentially rows) and macro-blocks of a video frame. In the temporal domain, the data flow can be sub-divided at a frame level or the group of picture (GOP) level.

Based on the granularity of the data access pattern we define 4 different frameworks.

1. Line processing (or slice with n=1)
2. Macro-block processing
3. Frame processing
4. GOP processing

If the data access pattern of an application can be determined to be one of the above four, then the corresponding framework can be easily used. There are also ways to integrate multiple frameworks for asymmetrical parallel processing (e.g. when you have two or more processing algorithms for a data stream). We show one example of how to modify the frameworks for such applications. We also show ways to modify the frameworks if dependencies extend beyond the 4 levels described (e.g. dependency between multiple macro-blocks in a motion window search etc.).

Template	Core cycles/pixel(approx.) single core	Core cycles/pixel (approx.) 2 cores	L1 data memory required	L2 data memory required
Line processing	42	82	(line size)*2; for ITU-656 1716*2	
Macro-block processing	36	72	(Macro-block size(nxm))*2	(macro-block height *line size)*4
Frame processing	35	69	(size of sub-processing block)*(number of dependent blocks)	(size of sub-processing block)*(number of dependent blocks)

Table 1: Specifications for the frameworks

We will discuss the data flow models for each of the framework. For example the line processing framework can be described as follows

Line processing

In case of line processing dependency exists only at a line level i.e. between adjacent pixels. Every line forms a data block which can be independently processed by each core. Figure 1 shows the data flow model for the line processing framework. The video input is handled by core A and video out is managed by core B. Separate sets of MDMA channels are used for managing data between core A and core B. Multiple buffers are used in L1 memory to avoid contention between core and peripheral-DMA access. Synchronization between the two cores is required every line which is achieved with a counting semaphore.

Examples of applications that can utilize this framework include color conversion, histogram equalization, filtering, sampling etc.

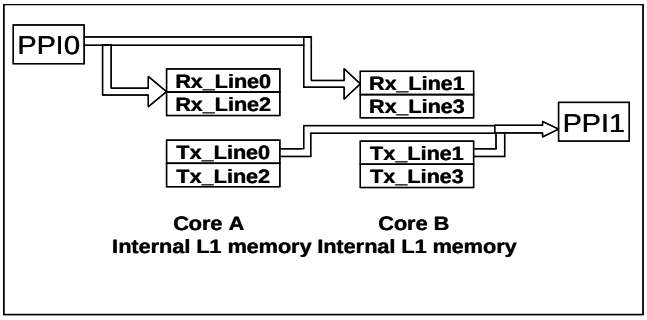

Figure 1: Data flow mode for the line processing framework

Framework analysis

Benchmarking

To evaluate the dual-core frameworks, we first develop a single core application with the data flow model and then compare it to the dual-core implementation. The single core models are discussed in [1] in more details. We compare the speed up of only the basic frameworks and not the combination of frameworks as described before.

The cycles shown are the core computation cycles available for processing the stream data to meet real time constrains for an NTSC video input. For a core running at 600MHz the total cycles available per pixel to meet the real time constrains 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. One can thus assume an infinite L1 memory for any core accesses to the stream data, when using the frameworks.

As can be seen in Table 1, the dual-core frameworks effectively give a 2x speed-up on all the frameworks. The table also shows the L1 memory usage for each core and the shared memory space required for each of the framework. The frameworks use the ADI provided Device Driver/System Services Library (DD/SSL) for peripheral and data management.

Conclusion

We have shown that by understanding the data access pattern of a particular application and effectively utilizing the memory and system resources of the underlying architecture a scalable parallel application can be developed.

References

[1] Kaushal Sanghai, "*Video Templates for developing multimedia applications on Blackfin processors,*" Application note, Analog Devices Inc, Sept. 2006.

[2] *ADSP-BF561 Blackfin Processor Hardware Reference*. Rev 3.1, May 2005. Analog Devices, Inc.

[3] David Katz and Rick Gentile. *Embedded Media Processing*. Newnes Publishers, Burlington, MA, USA, 2005.

[4] *Device Drivers and System Services Manual for Blackfin Processors.* Rev 2.0, March 2006. Analog Devices, Inc.