

Partitioning video processing algorithms onto multi-core architectures has been researched for decades, and over this time several techniques of varying efficiency have been developed to divide up the work among the processors. Let's take a closer look at some of these techniques, and see how video processing poses unique challenges to the multi-core processor.

Data Partitioning

One partitioning technique commonly used is called data partitioning, which relies on the ability of processing data blocks in parallel. This technique is most commonly and easily applied at a high-granularity, each data block being a channel, a frame, or a slice, where a slice refers to a large area of a frame that is processed independently from the rest of the frame. Each data block is processed in parallel on a different processor. A master processor is generally responsible for ensuring synchronization among the processors and combining the results as needed.

Applying this partitioning approach at a high-granularity presents the advantage of requiring only a minimal amount of inter-processor communication, since each block can be processed independently from the others. This approach is also easy to implement as only few modifications to the existing single-core oriented reference code are required in order to run in parallel on all processors and produce functional output. However, this technique has also inherent problems. First, it is difficult to ensure proper load balancing among processors as video codec algorithms have data-dependent processing requirements. For example, one slice of a frame (or video sequence) may contain scenes with small amount of movement and details (e.g., a uniform background such as a wall or the sky), resulting in much lower processing requirements than another slice containing high-motion scenes with finer details (e.g., the face of a person talking).

Because of these discrepancies, some processors remain idle for long periods of time while others are busy processing the most computationally intensive scenes: this unbalance translates as a waste of the processing resources and suboptimal performance. Second, simple data partitioning results in non-scalable implementations, since there is little flexibility on the number of blocks in which the data can be divided. For example, a 16-channel encoder may fit nicely on a 16-processor architecture by assigning one processor to each channel, but the code will need to be reworked significantly if another application needs to run in parallel on that architecture and mobilizes one or more processors for extended periods of time. Dividing frames into slices offer slightly more flexibility as the size and number of slices can generally be adjusted without requiring extensive code changes. Unfortunately, dividing a frame into too many slices deteriorates the efficiency of the compression algorithms.

Data Pipelining

Another partitioning technique is called data pipelining or functional partitioning. This technique consists of assigning each processor a different processing block, such as motion estimation or texture encoding in the case of video encoders. With this approach, the data is processed in a pipelined fashion: one processor applies the first processing block to the data and passes on its output to another processor, which applies the second processing block to the modified data, and so on. Using the same approach, the most challenging processing blocks can be subdivided and assigned to multiple processors while simpler ones can be handled by one processor alone to achieve better load balancing.

Unfortunately, this second partitioning technique also introduces multiple challenges. First, the assignment of each processing block to a processor is generally done at compile time: this is the simplest approach and sometimes the only approach that is possible because of the limitations of the architecture or the lack of a multi-core operating system running on the architecture. Assigning roles to each processor at compile-time does not allow for proper load balancing. For example, motion estimation processing requirements vary greatly depending on the video streams being processed: still and low-motion sequences result in lower processing requirements for the motion estimation block than high-motion ones.

As a result, processors assigned to the motion estimation algorithm are likely to be underused with low-motion scenes while they will be the bottlenecks with high-motion scenes. Second,

splitting an algorithm into multiple processing blocks run on different processors introduces inter-processor communications that may affect performance by straining memory resources or causing processors to stall as a result of data starvation. Finally, a simple data-pipelining approach suffers the same limitations that were mentioned about data partitioning earlier. Algorithms cannot be divided into an arbitrary number of blocks to match perfectly the number of processors available on a multi-core architecture, and assigning processing blocks to individual processors at compile time results in non-scalable implementations

Efficient Multi-Core Partitioning

Efficient partitioning of complex algorithms such as video encoders requires a combination of the two partitioning techniques described above, and the ability to assign tasks to processors at run time instead of compile time whenever appropriate. To overcome the limitations of data partitioning, the granularity of the blocks being processed by individual processors needs to be smaller than a slice, which introduces data dependencies that need to be dealt with. This granularity level may be at macroblock (MB) level or the level of a small group of MBs. Bringing down the granularity of data partitioning to a finer level and combining it with data pipelining creates a large number of individual tasks. This large number of tasks that can be allocated to processors at run time is the key to an efficient use of multi-core architecture resources.

Many challenges are in the way of this approach. How do you define tasks to minimize data dependencies? How do you decide in which order tasks need to be processed to ensure that there will always be new tasks available when a processor becomes available and despite the fact that the processing requirements of some of the tasks may vary drastically with the data being processed? How do you ensure that the task-switching overhead -- the time spent between when a processor completes a task and starts the next task -- remains small? How do you ensure that this partitioning approach is scalable so that you can assign a variable number of processors to one algorithm depending on the other algorithms running in parallel and the respective processing requirements? How do you ensure that each processor has enough fast memory available to process their tasks efficiently given that some tasks have much higher memory requirements than others and that the amount of fast memory is limited and shared across many processors?

The answers to many of these questions depend on the application being targeted, the multi-core architecture being used, and the software libraries and tools provided by the processor vendor to develop and debug code running on that architecture. In this article we focus on how Cradle implemented the MPEG-4 encoder on the CT3600 chip and provide elements of answers to many of these questions. We start with a brief overview of Cradle CT3600 architecture and the structure of video encoders like MPEG-4. We then discuss in detail how the MPEG-4 encoder was partitioned on the CT3600 architecture.

The CT3600 MDSP family

The Cradle CT3600 family of Multi-core DSP (MDSP) processors is a family of heterogeneous multi-core chips, accompanied by an easy-to-use multi-core programming system that comprises development, debug and profiling capabilities. One platform can be reprogrammed for any or all of the multi-channel, multi-application products.

The Cradle CT3600 architecture has up to 8 RISC processors and 16 DSPs. It is a shared data memory architecture with all elements having their own instruction memory and 32-bit wide register files. Cradle defines a group of 4 RISC processors as a Quad. Associated with a Quad are 8 DSPs, 128k bytes of shared data memory and nine 8-bit Programmable I/O Ports, each embedding a CPLD and state machine (*Figure 1*).

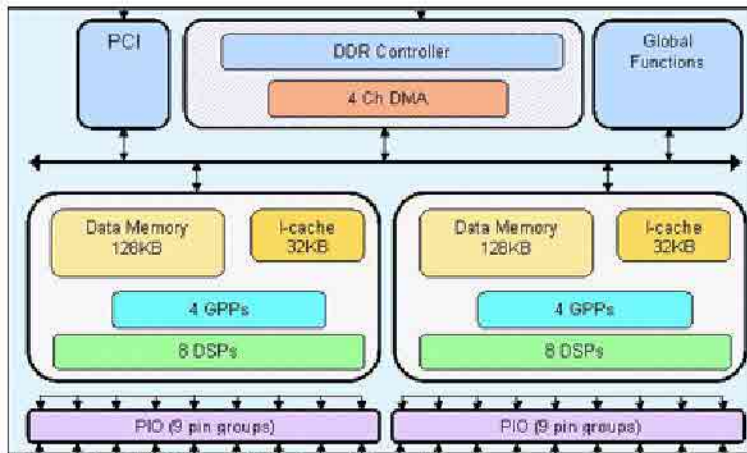


Figure 1: CT3616 architecture block diagram

Global resources include a PCI Bus interface and DDR-SDRAM controller with multiple DMA channels, Global Semaphores and bus-performance monitors.

Co-designed with the processor architecture is the Cradle SDK – a multi-core simulator and debugger Software Development Kit. All 24 processors and all I/Os can either be simulated or accessed in the hardware directly through a JTAG or PCI interface

MPEG-4 Encoder Structure

Video encoding algorithms like MPEG-4 typically consist of the blocks shown in Figure 2.

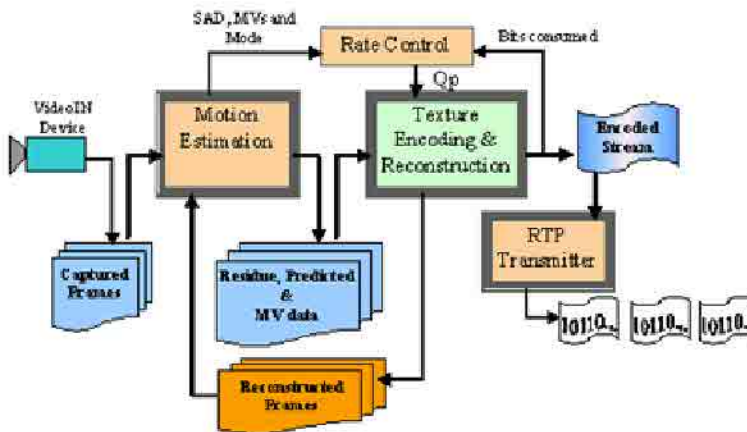


Figure 2: typical block decomposition of a video encoder

The two most processing-intensive blocks in the MPEG-4 encoder are Motion Estimation (ME) and Texture Encoding (TE). These blocks have different memory and processing requirements and exhibit different data dependencies.

The ME requires large amounts of local memory in order to hold the search area without stressing the DRAM bandwidth. The ME and TE have roughly comparable processing requirements, but both block processing requirements are data dependent. In the case of the ME, low motion translates into a smaller number of search iterations and fewer computations. The quality of the initial motion vectors used by the ME when processing a MB has a noticeable impact on the efficiency of the ME algorithm. Poor initial motion vectors generally lead to longer iterations and potentially poor final motion vectors. The best initial motion vectors are usually derived from neighbouring blocks on which the ME processing has already been applied. As a result, in order to obtain a high quality ME it is necessary to

process groups of MBs in a linear fashion, one at a time. Unfortunately, the larger these groups, the less processing that can be done in parallel since only one processor can be assigned to each of these groups. The size of these groups needs to be selected with care in order to obtain a fast, parallel implementation without compromising the compression efficiency of the codec.

Like the ME, the TE can operate on a single macroblock at a time. However, the TE has quite different characteristics. One of the most noticeable differences is that TE doesn't require as much local memory as the ME. TE includes a prediction mechanism that takes advantage of transform coefficients already computed for neighboring MBs in order to encode more efficiently coefficients belonging to the current MB being processed. This mechanism is commonly referred as AC/DC prediction. In order to take advantage of AC/DC prediction, coefficients from previous encoded blocks need to be kept in memory but they require only a fraction of the size needed to keep the large ME search area. The TE also includes a transform function that can be accelerated significantly when the input data contains a lot of zeros, which happens whenever a MB is predicted efficiently from another MB. Like the ME, the TE has some internal data dependencies: the TE can take advantage of an AC/DC coefficient prediction mechanism, which requires neighboring blocks to have already been processed by the TE.

However, because of the lower memory requirements of the TE, it is possible to run TE in parallel on multiple separate slices of a frame. Each slice is encoded separately and markers are inserted to identify the beginning and end of an encoded slice. The ability to work on separate slices in parallel makes TE a good fit for multi-core architectures, especially those able to support run-time allocation of tasks to processors since each slice may take a variable amount of time to be encoded. Having multiple slices per frame is also useful to recover from data corruption, since an error in the bit stream will not impact the entire frame but only a slice. The size of each slice results in a tradeoff between the amount of memory being consumed (since for each slice a buffer is needed to hold the coefficients from the previous row of MB), the quality of the encoding (since AC/DC prediction cannot be used to encode the first row of each slice), and performance (since the more slices we have, the more work can be done in parallel on multiple processors).

MPEG-4 Implementation on the CT3600

Now let's see how these procedures are mapped onto the CT3600 architecture.

Motion estimation partitioning

The memory requirements of the search area imposes some natural restrictions on the way ME needs to be partitioned on the CT3600 architecture. For example, processors working in parallel on the ME need to be assigned neighboring MBs in order to share the same search area and use the memory most efficiently. Neighboring groups of MBs on a same row need to be processed one at a time from left to right whenever possible in order to take advantage of the motion predictor vectors that were already computed for the closest MB. In the case of Cradle's implementation, each row of MB is divided in a handful of these groups, the exact number depending on the image resolution. The MB rows also need to be processed one row at a time in order to update the search area progressively using a circular buffer, and making only non-overlapping accesses to DRAM to minimize DRAM utilization. These various restrictions lead to a data partitioning approach for the ME procedure where a group of processors able to share the same local memory efficiently –processors belonging to the same Quad in the case of the CT3616 –work in parallel on contiguous groups of MBs belonging to the same row.

The amount of local memory that needs to be allocated for ME depends on the search area. For example, if the search range is set to 64 horizontally and 32 vertically, then the search area needs to contain five rows of macroblocks (80 pixel lines). The frame can be split into as many partitions as required based on the amount of local memory available. This is at the cost of increased bandwidth, since the overlapping search areas for vertical partitions need to be reloaded for every partition. In the current implementation, the number of vertical partitions is chosen dynamically during initialization given the amount of local memory ME can use.

The RISC controller for ME ensures that the search area and the current macroblock row are ready in local memory before the DSPs start working on a new macroblock row. The RISC processor uses the DMA controller to load both the search area and the current macroblock row in the background.

The motion vector data, the macroblock type, the motion compensated data, and other relevant information need to be communicated to the TE. This is accomplished by having each DSP store this information in the SDRAM using DMA transfers whenever appropriate.

Texture encoding

The TE works on data packets produced and stored by the ME in SDRAM. As previously explained, TE can be applied to multiple slices simultaneously since the local memory requirements for the TE are smaller.

The TE processing block itself is divided into two tasks following a functional partitioning approach. These tasks are the Pixel Processor Task (PPT) and the Entropy Coding Task (ECT). The PPT consists of computing DCT, Quantization and Inverse Quantization, Inverse DCT, and reconstruction. The PPT also stores the quantized DCT coefficients and other relevant information in SDRAM in what is called an ECT packet. The ECT processes the ECT packet and produces an MPEG-4 compliant bitstream. Each processor can be assigned to a PPT, which operates on an entire row of MBs, or an ECT, which operates on a slice. In the case of Cradle's implementation, a D1 frame is divided into four of these slices while a CIF frame maps directly into one slice. A PPT is pushed into the MTS PPT queue for every row of macroblock in the frame, and ECT is pushed into the MTS ECT queue for every slice of the frame. Several ECTs and PPTs can be processed in parallel provided that the relevant ECT packets are available.

These conditions allow for a dynamic partitioning approach where any DSP can process a new ECT or PPT as soon as the DSPs have completed the previous task they were assigned: this approach ensures proper load balancing since the large number of tasks keeps all DSPs busy continuously, thus taking full advantage of the processing resources available. This dynamic partitioning approach is implemented efficiently on the CT3600 by leveraging the Multi-Tasking Scheduler (MTS) tool that is part of Cradle's Software Development Kit: MTS provides software developers with a set of primitives for the RISC and DSP processors. The RISC processors act as controllers by defining, allocating, and controlling the order of execution of DSP tasks. The DSPs under the MTS framework run these tasks whenever they have completed the previous task that they were assigned. An important feature of MTS is that it offers a task switching mechanism with a low overhead, allowing work on relatively small tasks without incurring noticeable task switching overheads.

Implementation Validation

The partitioning of an algorithm such as an MPEG-4 encoder on a multi-core architecture results in a large number of tasks running on multiple processors in parallel. Because of the data dependencies that exist between these tasks, it is important to ensure that the resulting implementation runs efficiently and that no processor remains inactive for extended periods of time waiting for tasks to complete and data to become available. This validation step is critical for any multi-core implementation but can be challenging.

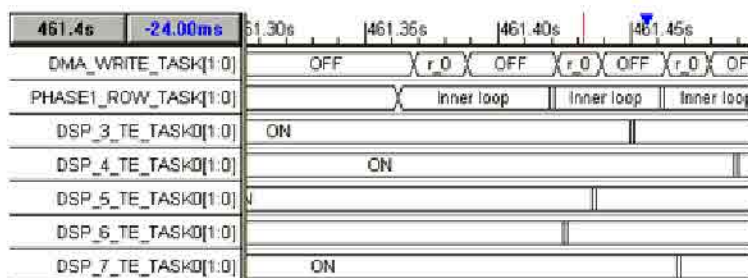


Figure 3: example view of processor activity obtained using Cradle's Run-Time Analysis Tool (RTA)

In the case of the CT3600, a number of sophisticated multi-core profiling tools are available to help understand the real-time activity of the processor resources being used. These tools can automatically generate various chronological (see the example in *Figure 3*) or average views of processor and memory resource utilization. Such tools are critical to fine-tune the definition of the tasks, the order in which they need to be processed, and to understand the impact of varying the frame resolution or number of channels that need to be processed.

Conclusion

A good partitioning takes into account the specificity of the algorithm being implemented and the target architecture. A flexible architecture allows the software developer to use a combination of data and functional partitioning approaches in order to realize the full potential of the architecture. A robust set of development and profiling tools is also key in helping the software developer with the implementation process and ensuring that all resources are used efficiently in the final implementation.

In the case of the MPEG-4 encoder implemented on the CT3600, an arbitrary number of processors within one of the quads is assigned to the ME processing block. The various restrictions imposed by the ME on memory utilization makes data partitioning with a compile-time processor allocation the most appropriate approach. The other computational-intensive processing block, the TE, has fewer restrictions and allows for a hybrid partitioning approach combining data and functional partitioning. Processors are allocated to TE tasks at run-time, which achieves processor load balancing by keeping all processors busy as long as there are frames to be processed, regardless of the type of video stream being processed. This approach leads to an efficient use of the computational resources, which can be shared with other applications running in parallel on the chip. The control and other non-compute-intensive tasks are handled by a small number of RISC processors. The resulting implementation is fully scalable, supporting a variable number of channels depending on the resolution and number of processors available.