Chapter 5

Video Encoder Implementation on Tilera's TILEPro64™ Multicore Processor

José Parera-Bermúdez, Javier Casajús-Quirós and Igor Arambasic

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/53429

1. Introduction

The Moore's law states that the transistor number on integrated circuits approximately doubles every two years. This trend has been met since its description in 1965. But this exponential growth in transistor count does not always translate into similar growth of CPU performance; some issues such as power density, total power and intra-chip distances are preventing clock speeds above 4.5 GHz. During the past decades advances in semiconductor technology and architecture have overcome the obstacles, but at present there is no alternative technology and all the possibilities of micro-parallelism seem to have been explored. Another major issue is that the speed of dynamic memory has not grown with the same strength as the CPU's speed, while static memory is prohibitively expensive for widespread use.

The solution being put into practice is the use of the so called multicore CPU, i.e. the integration of multiple cores on a single chip. Today nearly all computers, including desktops and laptops, are equipped with CPUs with at least 2 cores and it is not uncommon to see servers with 8 or 16 cores.

The evolution and the steady decrease in the price of technology have enabled the digital video to be a media component included on any device from small pocket players to professional projection equipment on movie theaters. Today the *de facto* standard for video coding is ITU-T/ISO H.264 /MPEG-4 Part 10 or AVC (Advanced Video Coding) [1]. Since its first publication, back in 2003, it has become one of the most commonly used formats due to its flexibility to be applied to a wide variety of applications on a wide variety of networks and systems, including low and high bit rates, low and high resolution video, broadcast, DVD

storage, RTP/IP packet networks, multimedia telephony systems, etc. In 2004 the standard was extended to enable higher quality video coding by adding several new features (increased sample bit depth precision, higher-resolution color information, adaptive switching between 4x4 and 8x integer transforms...) required by professional applications.

H.264 performs significantly better than any prior standard under a wide variety of circumstances in a wide variety of application environments, and outperforms MPEG-2 video, the DVD standard for movies, typically obtaining the same quality at half the bit rate or less, especially on high bit rate and high resolution situations.

Like other ITU-T standards, H.264 only specifies the syntax of the bitstream and the decoding procedures for reconstructing the video images; the encoding process is not specified at all allowing the use of different approaches, algorithms and optimizations as long as the bitstream is syntactically correct. Unlike previous standards, it is designed bearing in mind its implementation avoiding complex calculations and favoring the use of just adders and shifters; nevertheless encoding is far more involved than decoding.

It is easy to find lots of papers and books dealing with almost every aspect of H.264; there are also countless proprietary and open source software libraries and custom hardware implementations particularly for the consumer market. Therefore, what is special about the implementation described in the following paragraphs? In brief, the remarkable aspects are:

- It is targeted to very high quality with very low latency,
- It is a software-only solution, and
- The hardware is based on a commercial off-the-shelf multicore processor: the TILE*Pro*64™ from Tilera Corporation.

The performance achieved allows encoding 4K (*DLP Cinema Technology*, 4096x1716 pixels, 24 frames per second) video, the current standard for digital cinema, in real time with just one frame latency.

The study has been undertaken as part of a project that develops optimized hardware for those applications in which real time analysis-synthesis of high definition image streams is needed. The project was led by Datatech (www.datatech-sda.com) and it focused on a particular case of search & track applications for the aerospace segment, namely automatic refueling of flying military aircrafts.

2. System architecture

Figure 1 shows the hardware building blocks of the system. As can be seen the TILE processor is the very heart of the system and only some adapting logic is required to deal with the camera output; the I/O capabilities of the processor, including the Gigabyte Ethernet interface, make the rest.

http://dx.doi.org/10.5772/53429

From a logical point of view, the system behaves as a standalone RTSP (*Real-Time Streaming Protocol*) server [2] that packetizes the video encoded data for RTP delivering [3] over an IP network.

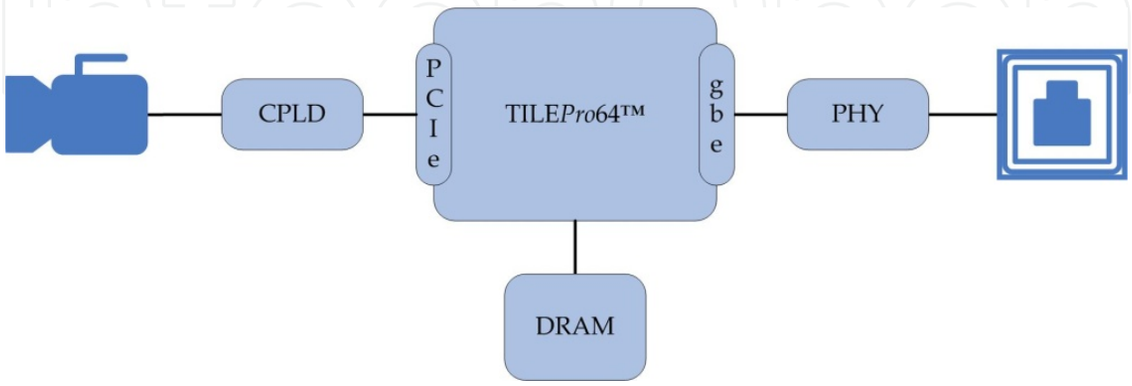

Figure 1. Hardware System Architecture

2.1. The TILE*Pro*64™ processor

The TILE*Pro*64™ [4], the second generation of Tilera's processors, is a fully programmable 64-core processor organized as a two-dimensional array (8x8) of processing elements (each referred to as a tile), connected through the iMesh™, a bunch of two-dimensional mesh networks. The processor also integrates external memory and I/O interfaces connected to the tiles via the iMesh™ interconnect fabric.

Each tile contains a Processor Engine, a Cache Engine, and a Switch Engine, which combine components to make a powerful, full-featured compute engine.

- The Processor Engine is a conventional 32 bit VLIW (Very Long Instruction Word) processor with three instructions per bundle and full memory management, protection, and OS support configuring a powerful, full-featured computing system that can independently run a Linux operating system. The Tile Processor includes special instructions to support commonly-used embedded operations in DSP, video and network packet processing, including: hashing and checksums, instructions to accelerate encryption, SIMD (Single Instruction Multiple Data) instructions for sub-word parallelism, saturating arithmetic, multiply-accumulate (MAC) instructions, sum of absolute differences (SAD), and unaligned access acceleration. All arithmetic instructions are of integer type because there is not a floating point unit.

- The Cache Engine contains the tile's Translation Lookaside Buffers (TLBs), caches, and cache sequencers. Each tile has 16KB L1 instruction cache; 8KB L1 data cache, and a 64KB unified L2 cache. This delivers a total of 5.5 MB of on chip memory. The cache can be configured as coherent or incoherent; in the first case, the hardware automatically maintains the consistency of data between processors, converting all on chip memory in a sort of L3 unified cache. Each tile also contains a DMA engine that works together with the cache engine for orchestrating memory data streaming between tiles and external memory, and among the tiles.
- The Switch Engine implements six independent networks. The Switch Engine switches scalar data between tiles through the Static Network (STN) with very low latency. Five dynamic networks (UDN, TDN, MDN, CDN and IDN) aid the Switch Engine by routing packet-based data among tiles, tile caches, external memory, and I/O controllers. Of the five dynamic networks, only the User Dynamic Network (UDN) is user-visible; the others are used to satisfy cache misses from external memory and other tiles, and for various system-related functions. The Static Network in addition to the five Dynamic Networks comprise the interconnect fabric of the Tilera iMesh™. The user does not explicitly need to manage these networks; rather they are used by the system software to efficiently implement the application-level API abstractions, such as user-generated inter-process socket-like streams.

It is noteworthy that all the cores are identical forming a homogeneous architecture that contrasts with other notable multicore processors such as the Cell Broadband Engine (from Sony, Toshiba and IBM) or the DaVinci from Texas Instruments. As a result, programming is easier, more portable and more easily scalable. Furthermore, the combination of cores and interconnecting network enable different kinds of parallelism: fine or coarse grain, shared-memory multithreading, message passing multitasking, etc. making the architecture suitable for a broad range of parallel problems.

The TILE*Pro*64™ supports the following primary external interfaces:

- Memory: four memory interface channels, each supporting 64-bit DDR2 DRAM up to 800 Mbps, for a peak total bandwidth of 25.6 GB/s. The memory controllers are on-chip.
- 10Gb Ethernet: Two full-duplex XAUI-based 10Gb ports with integrated MACs.
- PCIe: Two 4-lane PCI Express ports configurable as 4-lane, 2-lane or 1-lane (4x, 2x, 1x) with integrated MACs, supporting both root complex and endpoint modes.
- 10/100/1000 Ethernet: Two on-board RGMII 10/100/1000 Ethernet MACs.
- HPI: 16-bit host port interface.
- Flexible I/O: 64 bits of dedicated Flexible I/O for programmable I/O and interrupt support, with frequency up to 150 MHz and streaming capability.
- UART, I2C and SPI ROM.

http://dx.doi.org/10.5772/53429

Figure 2 shows a block diagram of the processor:

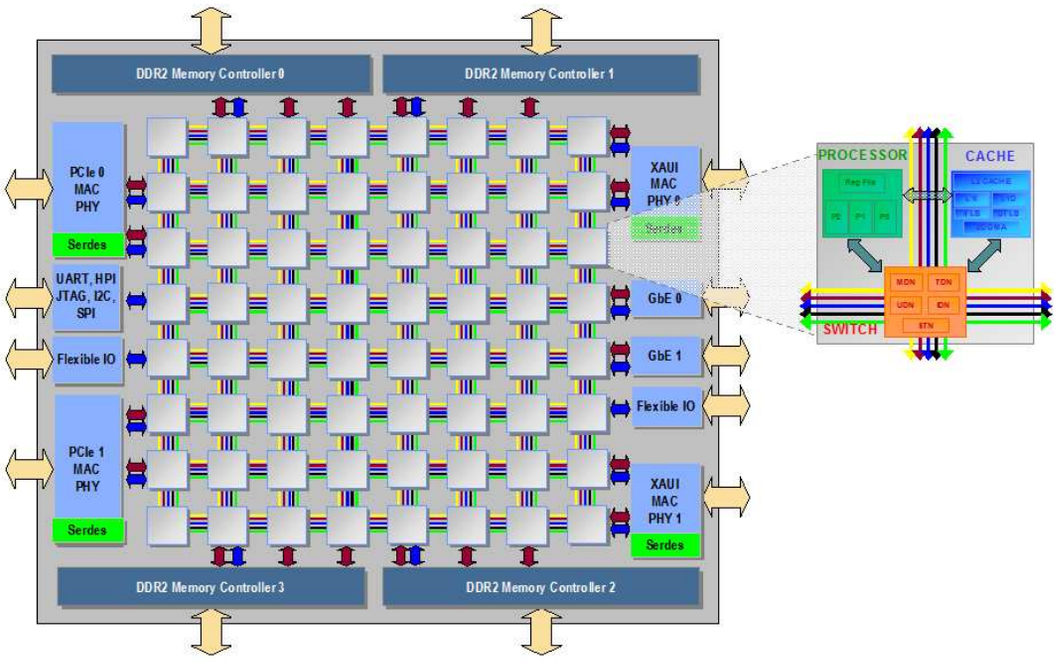

Figure 2. TILE*Pro*64™ Block Diagram

There are two versions of the processor that differ only in the operating frequency: 700 or 866 MHz. A few performance metrics at 866 MHz are listed in the next table:

Operations Per Second	8-bit	443 BOPs
	16-bit	222 BOPs
	32-bit	166 BOPs
Data I/O	40+ Gbps	
Memory I/O	205 Gbps	
Bisection Bandwidth	2660 Gbps	
On-chip Cache Memory Bandwidth	1774 Gbps	

Table 1. TILE*Pro*64™ Performance Metrics

2.2. The multicore development environment

The Tilera MDE [5] provides a complete software environment, including the system software stack, a variety of helpful software libraries, and standard Linux command-line utilities. The execution environment includes three layers: the hypervisor, the client operating system (Linux), and user space:

- The hypervisor is the lowest layer of the software stack. It abstracts hardware details of the processor, manages communication between tiles, from tiles to I/O controllers, and provides a low-level virtual-memory. This layer also provides I/O drivers that run on dedicated tiles and, therefore, do not run Linux or user space applications. The running drivers, the tiles on which they run and their parameters can be configured at boot time.
- The supervisor layer, composed of SMP Linux, provides system calls and I/O devices for user-space applications and libraries. This layer enables multi-process applications and multi-threaded processes to exploit multiple tiles for increased performance. The OS software manages hardware resources and provides higher-level services, such as processes and virtual memory allocation.
- The application layer runs user space programs that can invoke Linux system calls and link against standard libraries just as on any other Linux platform. Tilera provides the standard C/C++ run-time and other processor specific libraries.

The MDE also provides a complete suite of tools for all phases of program development, starting with authoring or porting an application, through debugging, and into performance evaluation. These tools include:

- C/C++ compiler, assembler and linker, and other standard Unix tools. This tool chain is compatible with that of GNU; specifically, the compiler supports ANSI C99 standard as well as GNU extensions. The tools enable the use of portable source code, easy to program and with the same concurrency support as that available for Intel x86 processors in a Linux box.
- A standard, open-source gdb debugger with support for the Tile Processor architecture.
- A software simulator that provides cycle-accurate profiling and tracing.
- A custom version of the open-source Eclipse IDE providing a GUI interface for all stages of program development: authoring, building, running, debugging and profiling.

3. Encoder parallelization

Programming a parallel application is not an easy job. The design space is enormous: different kinds of parallelism, data granularity, tools… The algorithm being implemented, the performance objectives and the computing platform impose some constraints but do not determine the design choices. Fortunately, the Tilera's platform supports a broad range of possibilities. See, for example [6], which explores several alternatives for the H.264 encoder.

http://dx.doi.org/10.5772/53429

The design of the application has completed all four typical stages established by best practices: task decomposition, assignment, orchestration and distribution [7]. The following paragraphs detail the course of action.

3.1. Task decomposition

3.1.1. H.264 encoder procedures

An H.264 encoder [8] [9] [10] consists of a few basic procedures; besides these, the standard defines a wide range of ancillary techniques, many of them optional, designed to provide enough flexibility to be applied to multiple scenarios. The encoder structure (see figure 3) does not differ substantially from that of other encoders but its many details and subtleties allow for a much more efficient compression.

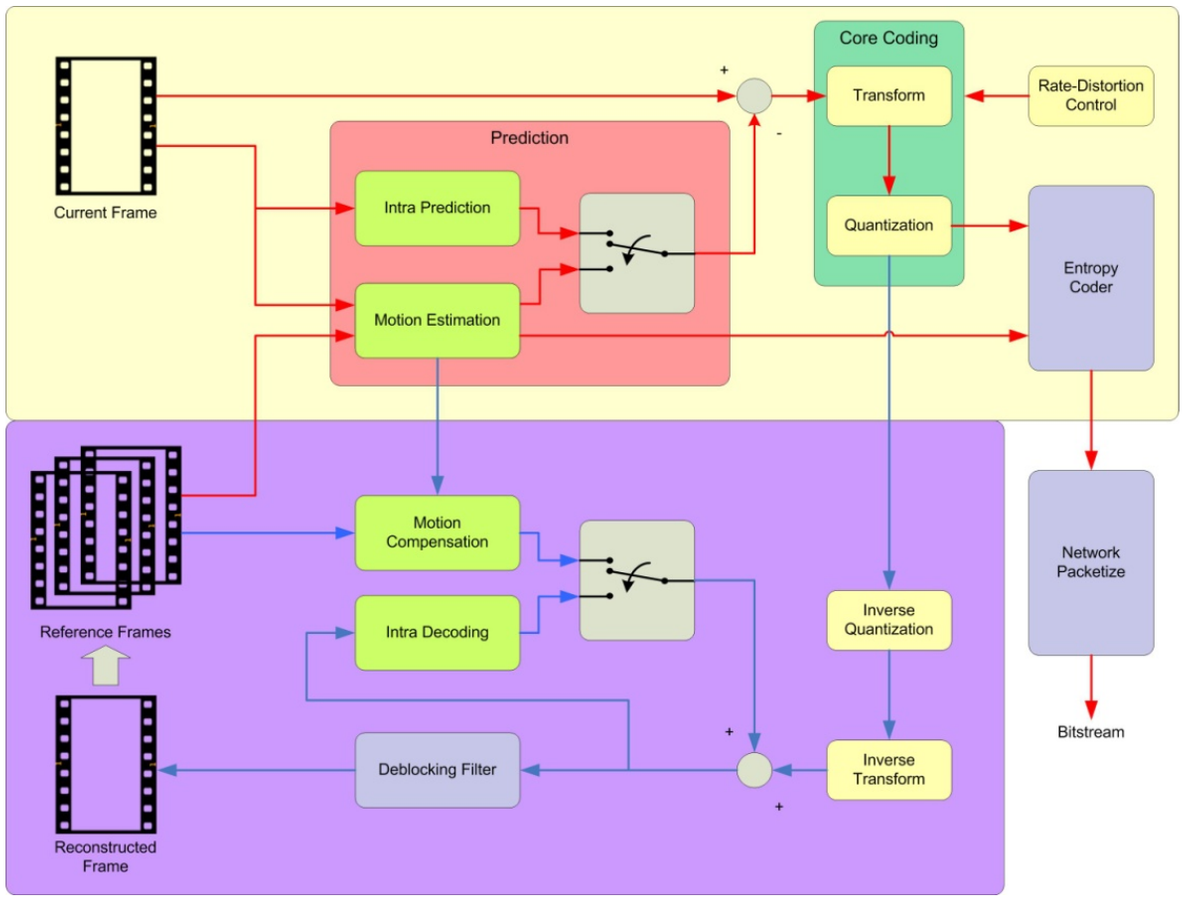

Figure 3. Block Diagram of Basic Encoder Procedures in H.264

The upper part of the figure (yellow) shows the encoding process. The frame being encoded, in YCbCr color space format, is divided into macroblocks, i.e. chunks of 16x16 luminance (Y) pixels and their corresponding chrominance (Cb, Cr) pixels, whose size varies according to the subsampling method (8x8 if 4:2:0, 8x16 if 4:2:2 or 16x16 if 4:4:4). The luminance and chrominance channels are processed separately using the same techniques.

In the prediction phase the encoder builds a macroblock using previously encoded data, either from the current frame (spatial or intra prediction) or from other frames (temporal or inter prediction). Intra prediction can be carried out for the whole macroblock in 4 modes or dividing it into 8x8 or 4x4 blocks in 9 modes. Each mode, related to a spatial direction, is just an extrapolation computed as averages of the neighboring pixels. Inter prediction is more involved since it tries to find a description of the macroblock by estimating its motion with respect to similar regions of previous frames. The search is performed on a variable number of reference frames, in several rectangular section sizes and with increased pixel accuracy to allow for sub pixel motion. Finally, the predicted macroblock can be expressed in terms of motions vectors from regions of the reference frames.

Once a macroblock is predicted the encoder subtracts the predicted pixel values from the input macroblock to form a residual. The prediction methods supported by H.264 make it possible to accurately predict the input macroblock, thus resulting in an outstanding video compression because, being a differential encoder, the residual values are small and very often nulls. Unfortunately, the computational complexity is too high, which poses a severe trouble for a real-time implementation.

The residual data is transformed using an approximate 4x4 or 8x8 2D DCT (Discrete Cosine Transform). This transform has the particular feature of compacting the energy of the input in the low frequency coefficients and thus the transformed residual usually contains a few non-zero values close to the matrix's upper left corner. H.264 does not use standard DCTs but modified so that their kernels consist solely of integer numbers eliminating the need for floating point calculations.

The modified DCT output should be quantized by dividing the coefficients by an integer, but the derivation of the transform left pending a scaling factor. Both numbers are combined to avoid division and, incidentally, to accommodate the quality parameter QP. The overall computation reduces the precision of the coefficients according to the desired quality governed by the value of QP: the larger the value, the poorer the quality but the higher the compression and vice versa. The rate-distortion control procedure can update the value of QP for each frame or macroblock in order to balance the opposing goals of high quality and low bit rate. Usually, the rate-distortion control procedure aims a maximum or constant bit rate but it is also possible to encode aiming constant quality in which case this procedure does nothing.

The quantized DCT coefficients are scanned in zigzag to sort them according to increasing spatial frequency; then they are converted into binary codes along with other signaling information (macroblock partitioning, prediction modes or motion vectors…) and inserted into the output bitstream. The binary codes are computed by the entropy coder procedure

http://dx.doi.org/10.5772/53429

using variable length or arithmetic coding, both adapted to the context to further reduce the number of bits. Hence, their names are context-adaptive variable length coding (CAVLC) and context-adaptive binary arithmetic coding (CABAC).

The quantization output also feeds the decoding process (the purple box in figure 3). This process is needed at the encoder because it reconstructs the macroblocks and frames to be used for prediction. The first procedure is the inverse quantization; actually it is a rescaling as quantization cannot be inverted. Afterwards the coefficients are inverse transformed to form a residual which is added to the prediction to get the reconstructed macroblock. If using inter prediction, the result could be optionally filtered for reducing blocking distortion by smoothing the macroblock edges. Note that typically the reconstructed macroblock will differ from the original due to the loss of precision caused by quantization. Ultimately, H. 264 is a lossy compressor.

3.1.2. Implementation tradeoffs

In order to fulfill the requirements of real-time, low-latency and high quality the encoder implements only a subset of the standard features and techniques available in the standard. This selection does not prevent the encoder to comply with the standard because H.264 allows a high degree of flexibility in the techniques used. Specifically, the implemented encoder has undergone the two following main tradeoffs:

- Intra only prediction, i.e. all predicted pixels are computed using only the current frame; otherwise, the very low latency requirement couldn't be achieved.
- CAVLC entropy coding; the alternative method, CABAC, is much more efficient from the standpoint of the bit rate but it cannot be parallelized due to its recursive nature.

These tradeoffs, and some others discussed below, adversely affect the compression ratio of the encoder resulting in an increased bit rate. Fortunately, neither the best compression ratio nor constant bit rate are requirements of the implementation. These goals are distinct from those required for consumer applications for which there are lot of solutions, but they are essential in many professional applications: remote monitoring, remote assistance, content generation, broadcast, video surveillance. The H.264 standard dedicates some specific profiles for this kind of applications (High, High10 and High10 Intra), that in some cases the industry has adopted, such as Panasonic's AVC-Intra.

Undoubtedly the temporal prediction improves the bit rate but not for free; in order to obtain a high quality a large number of reference frames are needed, scene changes can lead to devastating effects, especially if a constant bit rate is desired, and latency, measured from end to end, i.e. camera to monitor, increases linearly with the number of reference frames, and can reach 1 second, only for decoding.

In our implementation, the encoding of a frame begins as soon as the first 16 lines of pixels are available and once the row is encoded it is sent so that the decoding can start even before the whole frame is encoded. Such an extremely low latency is only possible using spatial prediction. Additional advantages of the intra-only prediction are: 1) ease of frame by

frame video editing, 2) the resilience against transmission errors, since an error affects only one frame, and 3) a significant saving of memory, which is especially important for very large resolutions.

3.1.3. Amdahl's law

The very first step in parallelizing an application is to determine if it is worth. Regardless of costs, running platform, software architecture and any other constraint a parallel application can run faster only a limited amount compared to its sequential version. Amdahl's law states that if P is the fraction of code that can be made parallel and $S = (1 - P)$ is the fraction not parallelizable then the maximum speed up that can be achieved by using N processors is

$$speedup(N) = \frac{1}{S + \frac{P}{N}} \tag{1}$$

The results of applying Amdahl's law to a given problem are just a rough approximation to reality but serve to get an estimate of the maximum parallel performance and to focus attention on potential bottlenecks and hot spots that can be found in the algorithm under development. So, the essential starting point in parallelization is to get an optimized sequential code, in order to determine the value of S.

The available literature dealing with H.264 encoding focuses on algorithmic description or performance improvement but usually forget to emphasize the inherently non-parallelizable part: the composition of the bitstream. Once the input raw video is encoded the resulting data must be packaged into NAL (Network Abstraction Layer) units which are byte aligned structures with header and trailing data. The data, known as RBSP (Raw Byte Sequence Payload), is written into the NAL units using a strict syntax in which the macroblock raster order must be preserved. The number of bits generated depends on the image, making it impossible to compose the NAL units without complying with the order. Furthermore, the Annex B of the standard states that RBSP data must be checked against patterns of bytes that can confuse the framing alignment while decoding. Those patterns must be disambiguated by byte stuffing the RBSP, i.e. inserting a fixed 0x03 byte each time they occur. Again, this procedure is neither predictable nor parallelizable.

The execution of the optimized sequential code on a Linux box equipped with an Intel Core2 Duo (T7700) CPU @ 2.40 GHz reveals that the fraction of time spent handling the composition of NAL units is 0.45%, yielding the value of $S = 0.0045$. The encoded frames per second (fps) for a 4096x1716 video are 0.75. These figures have been obtained without taking into account the input or output in order to accurately measure the time spent in the algorithm. Solving equation (1) for $N = 60$ processors results in a speed up of 47.4x that applied to the throughput gives 35.56 fps, enough for the digital cinema format (24 fps). A similar run on the Tilera platform @ 866 MHz yields $S = 0.015$ (1.5%) and a throughput of 0.67 fps. Solving again for $N = 60$ processors results in a speed up of 31.8x and a final throughput of 21.33 fps, less than the requirement for the above mentioned format. The different values of S are mainly due to the unequal facilities fitted in the CPUs for handling bytes and bit fields. The

http://dx.doi.org/10.5772/53429

result of this analysis indicates that the NAL unit management is clearly a hot spot in the code that could ruin the overall performance. Obviously, an optimization is needed in the Tilera side to fulfill the goal.

3.1.4. Data dependencies

No parallel program can be built without knowing the data dependencies that the algorithm imposes. As previously stated, the basic procedures of the encoder are the pixels prediction and the entropy encoding of the residuals; in our case, intra prediction and CAVLC. It is clear that the second procedure must follow the first one, since it's not feasible to encode any data without having calculated it. Aside from this obvious fact, an analysis of the H.264 encoder algorithm from the data flow standpoint shows:

- The input image is partitioned for processing into so called macroblocks, square chunks of 16x16 pixels.
- The macroblocks are processed in raster scan order, i.e. from left to right and from top to bottom.
- Each macroblock is predicted using some data from previously encoded macroblocks, specifically the boundary pixels of the upper, upper right and left macroblocks. The only exception to this rule is when the neighbouring macroblocks are not available; for example, the first macroblock of an image does not use any additional information because their neighbours do not exist.
- In order to compute the entropy encoding each macroblock needs a quantization parameter, QP. There is no provision to determine how a macroblock selects this parameter, but usually this job is entrusted to the block labelled as “Rate Distortion Control” in figure 3, because it affects the number of bits generated in the entropy encoder and ultimately the bit rate of the whole encoder. The quantization parameter can be seen as a quality parameter: the lower the value the better the quality but also the higher the bit rate. Our implementation allows to select between constant quality (fixed QP) and constant bit rate (adaptable QP), but for the ease of parallelizing the latter option is applied in a frame by frame basis.

In summary, the data dependencies at this algorithm level are the boundary pixels from neighboring macroblocks and a frame constant quantization parameter.

3.1.5. Tasks

After analyzing the extent of parallelization and data dependencies it is the time to analyze the tasks that make up the algorithm. Here we mean by task not the usual computing term but any procedure of the algorithm that could be accomplished in parallel.

The core encoding algorithm assuming the above mentioned tradeoffs could be described as a kind of streaming with frames as elements and macroblocks as the units of computation. At the system level there must be a single task that implements the RTSP service, waiting for

a client connection and then delivering the RTP packets. At the frame level the following tasks can be identified:

- Compute the Rate-Distortion procedure (usually known as RDO with the O meaning Optimization). The result is the quantization parameter to be applied to the frame.
- Open and initialize a Network Abstraction Layer (NAL) unit
- Read the raw input pixels of the frame.
- Encode the frame in macroblock chunks in raster scan order.
- Write the encoded data to the NAL unit.
- Close the NAL unit.
- Update RDO with the frame information.
- Deliver the NAL unit.

At the macroblock level, the task list is as follows:

- Read the macroblock raw input pixels.
- Get the boundary pixels from neighbouring macroblocks.
- Select the best prediction.
- Compute the residual error.
- Transform and quantize the residual error.
- Inverse transform and quantize the transformed data.
- Encode the residual transformed data using CAVLC.

Note that some tasks at macroblock level can be interspersed with those at the frame level, e.g. once a macroblock is CAVLC encoded the resulting data could be written to the NAL unit, and therefore there is no need of collecting the data from all macroblocks and writing it afterwards. The rearrangement of task order and the intermixing at the described or even finer levels broaden the parallelization options as long as the data dependencies are met.

Two potential hot spots can be found at the input and output of data. A digital cinema camera with a chroma sampling of 4:2:0 produces 10,543,104 bytes of raw video data per frame totaling more than 240 Mbytes/s. If we assume a compression ratio of 10, the total output bit rate will exceed 24 Mbytes/s. These figures are not unmanageable, but indicate that the input and output procedures should be treated with special care and, as far as possible, run them overlapped with the rest of tasks in the algorithm.

Another hot spot is concerned with prediction. The luminance part of each macroblock can be predicted in three pixel sizes: 16x16, the full macroblock, four 8x8 blocks or sixteen 4x4 blocks; the chrominance is always predicted in full size blocks (8x8 if using 4:2:0 chroma format) for each component. Each block is explored in several modes related to different spatial

http://dx.doi.org/10.5772/53429

directions: four modes for 16x16 luminance and chrominance and nine modes for the rest. Summing up all modes by iterating all the luminance prediction modes for each possible chrominance prediction mode yields a total search space of 736 combinations, each with its associated metric. The standard says nothing about how to compute these metrics and, therefore, how to select the best prediction mode for each block. There are two main approaches to assess this measure: 1) in the spatial domain, calculating the cumulative sum of absolute differences between actual and predicted pixels and (SAD); and 2) calculating the same sum but using the data in the DCT transformed domain (SATD). The latter provides, in general, better results but the quality or bit rate difference is not significant when the video resolution is high. By means of a test suite we have determined that using SAD instead of SATD on high-definition (HD) and above formats, the bit rate increases by only 1% whereas the computational load is 30% lower. Needless to say, the approach chosen is the use of SAD. Luckily, it also allows taking advantage of some of the more specific and powerful instructions of the TILE*Pro*64™ processor: the "sum of absolute difference" SIMD group.

In whatever case, these computations are very time consuming; note that the prediction of 4x4 and 8x8 blocks requires the reconstructed neighboring blocks since this will be the information available at the decoder. Therefore, once a mode is selected as best for a given block, it must be reconstructed emulating the decoder procedure in order for the neighbors to use its boundaries. This circumstance has promoted a lot of research over the last years [11] aimed to diminish the search space making available fast methods to "predict" the best predictor from among a substantially reduced set modes without compromising too much the bit rate. We have chosen for our implementation a simple yet effective fast mode decision algorithm called "Selective Intra Prediction" [12]. The key idea of this algorithm stems from the fact that the dominating direction of a bigger block is similar to that of a smaller block and therefore it is feasible to avoid the computation of the unlikely modes after the determination of the best 16x16 mode. The algorithm has been combined with the usual early-termination technique, but in spite of this the fraction of time dedicated to the selection of the predictor exceeds 60% after manually optimizing the code.

3.2. Task assignment

3.2.1. Parallel pattern

Previous sections have explored the opportunities for parallelism highlighting the hot spots of the encoder. Now it is time to choose the most appropriate type of parallelism and to logically organize the tasks.

The best parallelization pattern for achieving high throughput is the pipeline; if in addition its number of stages is not large latency can be low enough. However, a video encoder is not a good candidate for pipelining because, among other considerations, the computational burden of tasks is very dissimilar, the flow of control is not regular as it depends on data, data must be shared or copied and, specifically for the TILE processor, the number of stages should be no less than 60.

If we reject the pipeline approach, the remaining choices to consider are multiprocessing, multithreading or a mix of both. The main difference concerns virtual memory space; a process has its own non shared virtual space while a thread shares it with all other threads. Multithreading demands a more elaborated synchronization among threads but facilitates the inter-thread communications because it is accomplished simply by sharing data in memory. Furthermore, the TILE processor implements inter e intra-core cache coherence techniques that leverage the user of worrying about correctness of data. Based on these considerations the multithreading approach was chosen for the encoder.

3.2.2. Data decomposition

Another issue has to do with the decomposition of data, i.e. how to partition and distribute the data space among the cores. The encoding problem has not a recursive nature and so it is clearly preferable a geometric decomposition ideally suited to the data dependencies of the algorithm. It seems that macroblock decomposition of data is the right choice. Additionally, this decomposition enables the use of a single-program multiple-data (SPMD) model that eases programming by means of parametrizing the input to the code.

3.2.3. Core processing threads

So far we have decided to use threads to process macroblocks; the question now is how to organize those threads. A digital cinema video frame is composed of 27456 macroblocks; if any single thread is responsible for a single macroblock we would need the same amount of threads. Even if spread into 60 cores the number far exceeds the Linux threading facilities. A better choice is to partition the data not into macroblocks but into rows of macroblocks; this yields just 108 threads, a more manageable figure that not compromises the SPMD model.

Such a thread assignment can be still improved. Programs typically let the threads die once they have finished their work, but thread creation and termination has some overhead that can be obviated by recycling them. This is a simple technique, usually seen in digital signal processing, in which a thread created at startup runs forever until explicitly killed. For this technique to be useful it requires two synchronization points, the first to ensure that data is available while the second to signal that the work is finished. The exchange between thread management overhead and synchronization is worthwhile.

Further improvement arises if we avoid thread scheduling and time sharing in any single core as it eliminates the operating system kernel overhead devoted to task switching. In such a scenario each thread dynamically selects the next row to be processed as soon it has finished with the current. This technique, known as thread pooling, is especially well suited to the TILE architecture since the pool can be spread among the cores, each one running a single thread. The MDE has a provision for exploiting this setting: the so called dataplanes in which the standard Linux kernel is substituted by a zero overhead kernel. A nice result of using a thread pool is a fair load balancing, which is not always easy to get.

It remains to determine the scope of the row processing, i.e. which algorithm tasks the row threads perform. Referring to the above list of macroblock level tasks it is worthwhile that

http://dx.doi.org/10.5772/53429

the row threads be in charge of reading all the row input pixels, select the predictor, and so on, including the entropy encoding; the last task would be impossible if CABAC were used instead of CAVLC because CABAC, being recursive, needs data from the last macroblock of the previous row. In such a case the row threads could not proceed in parallel or, if they did, they should store all the information of the macroblocks and afterwards apply the entropy encoding. This would represent a severe waste of memory and an unmanageable hot spot. However, the CAVLC encoding so partitioned also has a drawback: since the bit alignment of the row data in the NAL unit is unknown, the whole unit must be realigned. Anyway, the entropy encoding in parallel is advantageous.

The preceding paragraphs have focused on analyzing the processing of macroblock rows but we have said anything about the issue described in Section 3.1.2 concerning Amdahl's Law: the realignment and byte stuffing of CAVLC encoded data are limiting factors of parallel performance that may saturate the speed up. The implementation dedicates a core, known as framer, running a single thread of manually optimized assembler code to address this problem.

Figure 4 shows a simplified time line of 12 row threads working in parallel. It can be seen that the whole process resembles a macroblock pipeline, although technically speaking it is not. Some details are worth being described:

- All time intervals are sketched alike; actually, times depend on input data.
- Each row, except the first, must start with a delay at least twice the handling time of a single macroblock to ensure that the boundary pixels are available.
- The total time spent at any frame is much greater than the time required for packetizing its encoded data into a NAL unit, as expected for a pipelined structure.
- The video input is not sorted in a natural way, e.g. row 0 at frame N (the green one) needs data before row 11 at frame N–1 (the blue one). The input procedure must take this fact into account and allow for a non ordered access to the data.
- At any moment in time multiple frames may be simultaneously being processed; the worst case arises at the frame boundaries (T0 in the figure). It is not difficult to prove that if we assume a constant, say mean, macroblock processing time value the number of simultaneous frames can be as great as:

$$n = \left\lceil \frac{2(C - 1) + R}{R} \right\rceil \tag{2}$$

being C the number of macroblocks in a row and R the number of rows. Furthermore, the row threads could start working with the frame N+1 before the framing thread has had an opportunity to evict the row data of frame N. A simple n-buffer strategy at the row thread output is enough to solve this trouble.

- The framing proceeds in bursts at the beginning of a frame because the encoded data is available but afterwards it must wait for the rows to terminate. This drawback puts the framer thread even more under pressure as there may be time intervals during which it cannot perform any work.

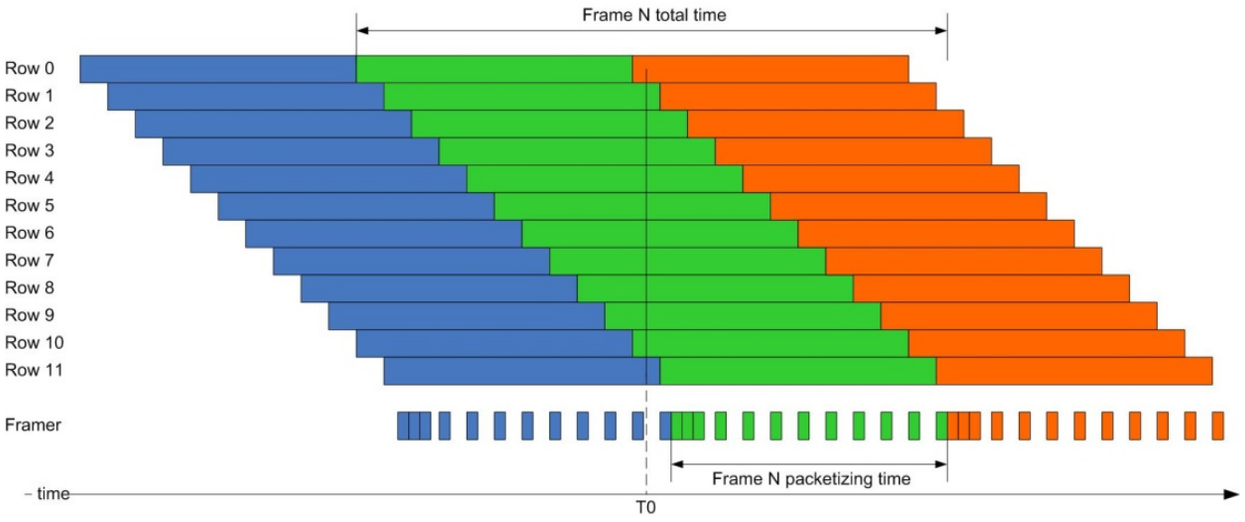

Figure 4. Time Line of Row Processing

3.2.4. Auxiliary threads

There are two tasks that still remain to be assigned to threads: the RTSP service and the RDO. The RTSP service can advantageously be implemented using two threads: the first one devoted to the service itself while the second in charge of the subsidiary real-time control protocol (RTCP). No one of these threads requires a great amount of CPU resources but the logical division facilitates software coding.

With regard to RDO, it could be as complex as desired in order to obtain accurate estimates of the bit rate and so select the best quantization parameter QP. But we have said repeatedly that optimizing the bit rate is not a priority of our implementation and a simple and fast PID (proportional integral derivative) controller algorithm is enough for our purposes. The only remarkable question is that adjusting the algorithm parameters should be made taking into account that input data are delayed due to the pipelined behavior of the processing. This RDO computation could be performed by the framer thread but we preferred to do it in a separate thread for the system to be more flexible in case of need.

3.2.5. Input and output

The best solution for input and output is that their functionality runs into two separate hypervisor drivers. Doing so, all I/O data could flow over the I/O Dynamic Network (IDN) that connects all tiles with the on-chip devices alleviating the burden of memory sharing at the user level. In addition, this schema obviates the intermediate level of buffering needed between the program and the Linux kernel drivers.

The output driver is just a packet based service tailored to the handling of the RTP payload over IP. A notable optimization feature is that the driver uses only fixed size buffers that fit into the Ethernet jumbo packets with two objectives: to reduce overhead and to avoid IP fragmentation.

http://dx.doi.org/10.5772/53429

On the other hand, the input driver is programmed as a server that handles the necessary buffering for extracting and reordering the camera data and delivers it at the pace enforced by the row threads requests.

3.3. Orchestration

The aim of the orchestration phase is to design the mechanisms that will ensure the proper synchronization among threads; i.e. that all control and data dependencies are met. Section 3.1.3 dealt with data dependencies; the control dependencies arise from the task assignment to threads, specifically to take advantage of the always live thread pattern.

The synchronization primitives used are those provided by the POSIX 1b and 1c extensions, available in any Linux box and in the Tilera's software stack. The use of these primitives is easier than programming custom ones, although its performance is not always the best; but having selected a coarse data grain for the implementation their impact is very limited.

In essence, we use semaphores for synchronizing threads and read-write locks to protect the macroblock boundary data. The use of the latter instead of the usual mutexes allows a higher degree of parallelism as it does not blocks any reading thread if the writer has not acquired the lock. Bearing in mind that we have designed the assignment of tasks to threads so that there is only one thread on each core, the read-write locks can have spin flavor to avoid putting the threads to sleep while waiting for the lock.

3.4. Distribution

So far we have always used 60 as the number of cores dedicated to encoding but the Tilera processor has 64; let see why. On the one hand, the framer thread, being the major potential bottleneck of the system, claims a core for itself; on the other hand, for the input and output hypervisor drivers to do their work overlapped with the algorithmic computation they each need a dedicated core. The remaining core hosts all the other auxiliary threads: RTSP server, RTCP, RDO and the C main thread that just waits for the program to exit.

The last issue to be addressed is how to distribute the threads, i.e. to determine in which physical cores they will run. The best way to make the distribution is keeping as far as possible the data locality since the latency for accessing an adjacent core's cache memory is much cheaper than accessing any other core's cache. This arrangement is easily attained for the row processing cores; unfortunately, there is no way for the framer core to be adjacent to all row processing cores. The selected distribution is shown in figure 5, in which row cores are shown in blue with the closed adjacency path in light blue. The framer core is shown in dark blue, input and output cores in orange and the auxiliary core in green. The position of input and output deserves a comment; they are physically very close to their corresponding hardware as Tilera recommends.

This distribution scales almost linearly for any video resolution with at least 60 rows of macroblocks, i.e. 960 pixels high, including high-definition (1080 pixels, 68 rows) and above. With lower resolutions the row cores will not all be active so there will be a degradation of performance.

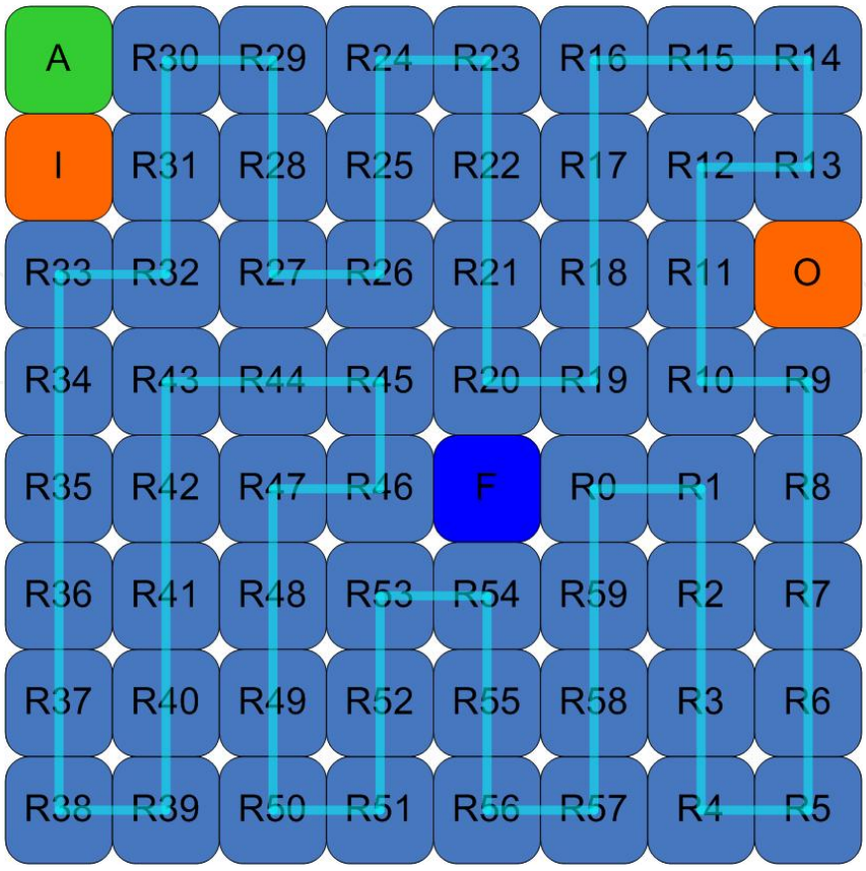

Figure 5. Distribution of Threads into Processor Cores

4. Results

In order to evaluate the results the freely distributable test video sequences Park-Joy has been encoded in different sizes using a constant quantization parameter QP = 18. This figure allows an encoding without noticeable visual degradation. Park-Joy contains small figures of running people; sometimes large objects - unfocused trees near the camera - move to the left as a result of a strictly horizontal camera movement, overlapping the entire scene. At the end of the sequence the camera slows the motion.

In sequential runs the mean time spent in macroblock encoding is 40.27 μs in Linux and 79.54 μs on the TILE processor @ 866 MHz. The differences are due to the operating clock and the architectural dissimilarities. It is easy to see that the optimizations undertaken in the Tilera side have been successful since the clock speed is reduced by a factor of 2.77 while the time ratio is only 1.98. Note that the Linux code has been optimized only at the C level and thus not using the SIMD instructions provided by the MMX or SSE instruction extensions.

The same run on the Tilera's simulator in functional mode in, which the cache hazards are not fully considered, yields 57.07 μs. It is apparent that the TILE core cache memory is not large enough to hold all code and data and thus incurring in a high rate of capacity misses.

http://dx.doi.org/10.5772/53429

In parallel runs the cache problem becomes more evident, as shows the following table:

Image Size	Simulator	Hardware
1280x720	239,82 fps	181,65 fps
1920x1080	155,98 fps	99,44 fps
3840x2160	37,37 fps	25,66 fps

Table 2. Encoded Frames per Second

It is worth to mention that the performance boosts around 32.5% in mean by avoiding the 8x8 block encoding of luma. This figure puts a little more spice to the controversy over the inclusion of this technique in the standard.

The following graph shows the throughput measured as time per macroblock (blue) and number of macroblocks per second (green) versus resolution.

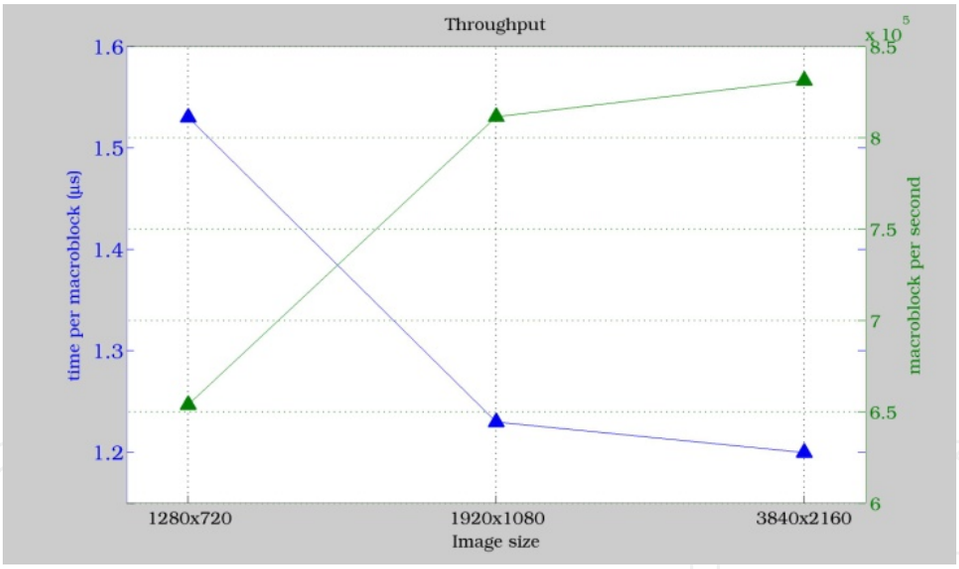

Figure 6. Throughput

It can be seen that the throughput degrades abruptly when the number of used row cores is less than available; the 1280x720 uses 45 cores, while the other uses all 60.

The next graph shows the speed up as a function of the number of row cores. The shape of the graph is quite linear but the slope is less than 1 as predicted by Amdahl's Law.

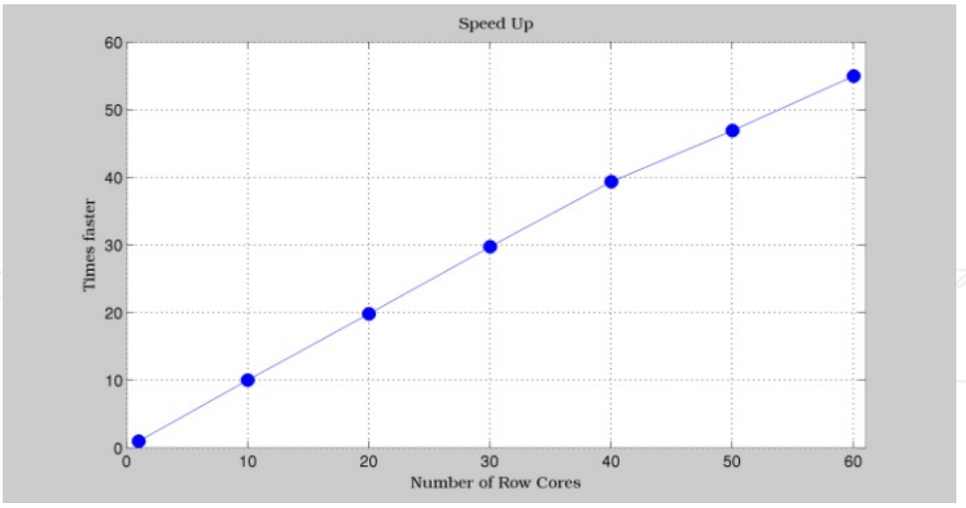

Figure 7. Speed Up

4.1. Some TILE*Pro*64™ troubles

Despite the enormous amount of silicon and functionality provided by the TILE*Pro*64™ processor, some flaws have been detected:

- It is quite hard to optimize the code using intrinsics or assembler; it would be nice if the documentation [13] contain examples, tips and tricks.
- The processor instruction set architecture contains basic instructions for bit and byte rearrangement at the register level; these include the "byte exchange", "byte/word interleave", and "masked merge word" instructions. However, it lacks bit-field extract and insert and byte/word shuffle instructions. These capabilities are incorporated in the new generation of Tilera processors, the TILE-Gx series.
- Correct use of branches is difficult, even for the compiler; branch mispredictions result in pipeline hazards that increase instruction latency. Fortunately, the feedback based optimization technique [14] alleviates this issue but it is cumbersome when optimizing source code.
- Finally, the most important limitation for video encoding is the amount of cache per core; 64 Kbytes of L2 is not enough for code and data, leading to many cache-capacity misses and therefore many stalled cycles. The TILE-Gx series has 256 Kbytes of L2 cache per core, without any doubt a must for achieving better video encoding performance.

5. The future of video coding

Video coding technology will not stop at H.264. A new draft standard known as HEVC (aka H.265 and MPEG-H Part 2) is still under development. It features important improvements over H.264 centered on achieving bit rate reductions of about 50% and supports a wider

http://dx.doi.org/10.5772/53429

range of high definition resolutions. Computational complexity consequently increases by an estimated factor of two to ten, and maybe more.

The techniques by which those enhancements are realized should be analyzed from the point of view of our implementation.

As we mentioned before, the CABAC procedure as defined in H.264 is not amenable to parallelization. In HEVC special care has been taken to reduce data dependencies in the particular version of CABAC it implements. However, data dependencies have not disappeared, and this poses severe problems – albeit less so – for implementation on parallel processors.

In order to ameliorate spatial prediction, new modes have been defined in HEVC. In particular the number of modes is 35 versus 9 in H.264. We have already said that 60% computation time is taken by analysis and selection of the optimal encoding mode. Therefore one must expect a considerable increase in computation needs due simply to the number of prediction modes that must be explored; more even so, given the fact that the image is divided not in uniform macroblocks but in coding tree blocks (or coding units CU) with an inner structure of variable sizes of their own (64x64, 32x32, 16x16).

6. Conclusion

The overall performance of Tilera's TILE*Pro*64™ can be said to be outstanding for video coding applications. In the particular case of low-latency H.264 encoding the largest difficulties arises for the highest resolution values of the video stream. For these, large amounts of memory are required, exceeding what is readily available within the processor; the main limitation resulting from the relatively small cache memory. Notwithstanding this fact, we have been able to find memory management schemes and workarounds that make real time encoding possible even at the highest resolutions (4096x1716, 24 fps) contemplated in this work.

For other video codec applications, expectations are high. Inter frame prediction can probably be traded off by lower resolution values. The main conclusion being that the processor architecture is adequate for established coders, whose bases were laid a few years ago and which are still the subject of implementation research.

New developments such as HVEC hold enormous promise, but the difficulties surrounding real-time implementation are challenging, to say the least. It is likely that several years of research are needed to significantly advance in that direction. Of course this raises the question whether the architecture will be up to the coding schemes under development and/or what enhancements will be necessary.

The results of this work do not stop at video coding. Applications to novel fields such as virtual advertising and augmented reality in medicine are under study for current and future projects.

Acknowledgements

The. authors gratefully acknowledge the support provided by project IDI-20100823 of Spanish Government's *Ministerio de Economía y Competitividad*, under the leadership of Datatech SDA who also acknowledges that support. Project TEC2009-14219-C03-01 also provided support for this work.

The authors also acknowledge the continuing support and cooperation of Datatech SDA for ongoing developments of Tilera's processor capabilities: real-time video analysis, virtual advertising and augmented reality in medicine.

Author details

José Parera-Bermúdez, Javier Casajús-Quirós and Igor Arambasic

*Address all correspondence to: jose.parera@upm.es

Department of Signals, Systems and Radiocommunications, Polytechnic University of Madrid, Spain

References

[1] ITU-T Rec. H.264 | ISO/IEC 14496-10 version 16, *Advanced video coding for generic audiovisual services*, January 2012. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.264-201201-I!!PDF-E&type=items (accessed 9 September 2012)

[2] RFC2326, *Real Time Streaming Protocol (RTSP)*, IETF, 1998. http://datatracker.ietf.org/doc/rfc2326/ (accessed 9 September 2012)

[3] RFC6184, *RTP Payload Format for H.264 Video*, IETF, 2011. http://datatracker.ietf.org/doc/rfc6184/ (accessed 9 September 2012)

[4] Tilera Corporation, *TILE Processor Architecture Overview for the TILEPro Series*, 2009.

[5] Tilera Corporation, *Multicore Development Environment: Programming the TILE Processor*, 2009.

[6] Takeuchi, Y., Nakata, Y., Kawaguchi, H. & Yoshimoto, M. *Scalable parallel processing for H.264 encoding application to multi/many-core processor*. International Conference on Intelligent Control and Information Processing (ICICIP), August 13-15, 2010, Dalian, China. doi: 10.1109/ICICIP.2010.5565292

[7] Gove D. *Multicore Application Programming: for Windows, Linux and Oracle Solaris*, Addison-Wesley, 2011.

[8] Richardson I. *The H.264 Advanced Video Compression Standard, Second Edition*, John Wiley & Sons, 2010.

[9] Wiegand T. & Sullivan G.J. *Overview of the H.264/AVC Video Coding Standard*, IEEE Transactions on Circuits and Systems for Video Technology 2003;13(7):560-576. doi: 10.1109/TCSVT.2003.815165

[10] Sullivan G.J., Topiwala P. & Luthra A. *The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions*, SPIE Conference on Applications of Digital Image Processing XXVII, Special Session on Advances in the New Emerging Standard: H.264/AVC, 2004. doi: 10.1117/12.564457

[11] Milani S. *Spatial prediction in the H.264/AVC FRExt coder and its optimization,* In: Miron S. (ed.) *Signal Processing*, Rijeka: InTech; 2010. http://www.intechopen.com/books/signal-processing/spatial-prediction-in-the-h-264-avc-frext-coder-and-its-optimization (accessed 9 September 2012)

[12] Park J.S. & Song, H.J. *Selective Intra Prediction Mode Decision for H.264/AVC Encoders*, World Academy of Science, Engineering and Technology 13, 2008. http://www.waset.org/journals/waset/v13/v13-104.pdf (accessed 9 September 2012)

[13] Tilera Corporation, *User Architecture Manual*, 2010.

[14] Tilera Corporation, *Optimization Guide*, 2010.

IntechOpen