Implementing H.264 encoding algorithms

Frank Lee describes how algorithms can be implemented on a software configurable processor.

The significant coding efficiency of H.264 video compression technology opens a wide range of new applications for streaming video over a variety of media. This international standard has risen in importance with its recent adoption by 3G, DVD Forum, and DVB, joining MPEG-2 as one of the world’s most common digital video formats. The H.264 standard provides advanced algorithms for motion estimation, inter-prediction, spatial intra prediction and transforms. The H.264 standard supports motion estimation on blocks from 16x16 to 4x4 pixels. Residual data transforms are executed on 4x4 blocks with modified integer discrete cosine transform (DCT) which avoids rounding errors. In common with other standards, such as MPEG, the H.264 codec implementation is not explicitly defined. The standard defines the syntax of the encoded bit stream and the method for decoding the bitstream. Developers of H.264 require a development methodology that enables experimentation and refinements of their algorithms and the ability to deliver a real-time encoding of the video.

Keeping pace with Moore’s law, today’s CPUs continue to be massively powerful, but their architectures are not well suited for video processing. One approach has been to augment the CPU with hardware acceleration units called intrinsic instructions (such as Intel’s MMX/SSE2 and AMD’s 3DNow extensions). Acceleration hardware can be designed to support the block-based and pixel-level processing tasks that are not efficiently handled by the CPU architecture.

However, many of the core encoding tasks, such as motion estimation etc., which consume many CPU cycles, are also very dataflow intensive, thus requiring deep register pipelines with fast memory accesses. Traditionally this has been best met with a purely hardware approach. This paper will describe the design methodology and the results of using a single 300MHz software-configurable processor to achieve H.264 encoding of Standard Definition (SD) video at 30 fps.

An essential element of the recipe for success for real-time video encoding applications is delivering the best image quality that is feasible for a particular screen resolution, given real-world operating constraints. For example, an uncompressed D1 video stream has an image size of 720 x 480 pixels and requires 1.5 bytes for color per pixel. Such a stream has 518 kilobytes per frame, and at 30 frames per second consumes an impressive — and equally impractical — 15.5 megabytes per second storage and bandwidth.

For network-based applications, bandwidth is only one of the limiting factors for video. The larger a video stream, the more opportunities there are for packets to be delayed and therefore become unusable. Additionally, available network pipes need to peacefully coexist with other real-time streams such as voice data. Developers can reduce the bandwidth of a stream by reducing screen resolution, but in many cases, the loss of image quality is unacceptable in the marketplace.

Lossy compression algorithms provide the most cost-effective means for lowering bandwidth while retaining quality. Successive versions of the MPEG video standard have continued to lower bit rate while retaining more image quality through increasingly complex algorithms. H.264, also known as MPEG-4 part 10, was designed to facilitate video transport over IP networks, and provides a substantial compression ratio over MPEG-2 while delivering equivalent or better image quality. In order to get this additional compression requires additional tools, new algorithms and more computations.

From a technical standpoint, the primary difference between H.264 and its predecessors is the use of multiple reference frames, wider search ranges, and smaller macro blocks for motion estimation which ultimately translates to computational intensiveness. Figure 1 shows the architecture overview of an H.264 encoder. The efficiency of the encoder can be found in the implementation of the following functions:

- Intra-prediction utilizing Forward and Inverse Discrete Cosine Transforms (DCT),
as well as Forward and Inverse Quantization

- De-blocking filtering
- Motion Estimation employing inter-frame comparisons to _PEL and _PEL accuracy.

Each of these functions requires extensive processing and must be performed in real-time to be useful. The remainder of this paper will focus on how software-configurable processors provide a development methodology for creating cost-effective H.264 implementations that enable experimentation and refinement of algorithms.

The need for hardware acceleration

Clearly, a task as compute intensive as H.264 requires hardware acceleration. OEMs provide significant value in how they implement functions such as motion estimation, and a programmable platform is required to enable developers to continue to refine their algorithms and sharpen their competitive edge.

A single traditional programmable processor has not provided the necessary performance or compute resources to implement real-time encoding at the highest level. And for base implementations, the development requires hand-optimized assembly language, a time consuming, architecture specific implementation.

ASIC implementations, at this stage of H.264 adoption, are inflexible. Implementing proprietary algorithms in programmable logic appears to provide the flexibility to address the evolving algorithm. These types of implementations typically have a programmable processor to handle application-level tasks and manage the flow of data to the programmable logic devices. The challenges of these implementations are many. First, there are a variety of ways to interconnect the processor and logic devices, such as an FPGA. The choice of the interface mechanics affects the throughput and thus the efficiency of the multi-chip solution.

The interfaces can be defined to operate synchronous or asynchronous. Synchronous operation burdens the hardware designer to meet timing requirements, while asynchronous operation introduces significant latency forcing stalls in the processor to complete the data interchange. The asynchronous interface can often expose limitation in the hardware handshake interfaces which may further reduce the effective data bandwidth. Using discrete devices as illustrated in figure 2, the CPU must prepare data and hand it off to the FPGA coprocessor unit.

The CPU must then wait for the FPGA unit to make the computation. Even though the units may appear to operate in parallel, the interdependency weakens the effect of pipelining of operations. Second, the CPU plus FPGA architecture has two development environments, further complicating the design. Often the FPGA architecture is based on a software implementation of a working solution failing to meet the performance requirements. The hardware team accepts the software algorithm then recodes the computational blocks in a hardware description language (HDL) and uses a separate verification methodology. Finally, in order to change the software algorithm or the context in the application, the FPGA coprocessor must also be changed. For an evolving algorithm, this issue can significantly impact the flexibility to evolve the algorithm and the delivery of a timely solution to the market place.

Software-configurable processors address these issues by integrating programmable logic within the processor datapath (figure 2), enabling the efficient handoff of data, a single development language and efficient changes to the software algorithm under control of the algorithm developer (figure 3). With the Stretch software-configurable architecture, entire functions written in C/C++ are compiled by the Stretch C compiler to form extension instructions that reside in the programmable fabric.

Using a single design environment, accelerating the compute intensive functions of an algorithm is accomplished at software compile-time. There is no need to recode the function in assembly language nor hand-off the functioning C/C++ code to a hardware engineer to redesign, rewrite and add a complex processor interface to the logic.

The Stretch S5000 family of software-configurable processors is based on the S5 compute engine that combines a RISC processor with programmable logic resources,
or Instruction Set Extension Fabric (ISEF), within the processor’s datapath. These instructions are accessed through the processor’s instruction pipeline just like any other instruction, making custom instructions equivalent to the processor’s native instruction set. A 128-bit wide register set facilitates efficient passing of large amounts of data and context to the ISEF so that multiple data can be processed in parallel.

The integration of extension instructions provides a substantial increase in the performance of complex algorithms. The S5 compute engine, operating at 300MHz, without extension instructions operates like a normal 300MHz RISC processor; however, with the addition of extensions the processor outperforms multi-GHz processors. As an example, the Stretch software-configurable processor out-of-the-box performs the EEMBC Telemark benchmarks with a score of 4.6. With the addition of extension instructions, taken from the original C/C++, the software-configurable processor score climbs to 877, outperforming assembly language optimized multi-GHz processors.

**Implementing H.264**

The algorithms used in H.264 are prime candidates for flexible hardware acceleration because of the amount of computation required. Additional acceleration can be achieved by taking advantage of the inherent parallelism in these algorithms.

Consider the processing of 4x4 blocks of luma pixels through a 2-D discrete cosine transform (DCT) and quantization step. The H.264 DCT and quantization matrices are shown in figure 4. The computations for the DCT portion of the matrix computation can be reduced to 64 add and subtract operations by taking advantage of symmetry and common sub-expressions. All 64 operations can be combined into a single ISEF instruction. Quantization (Q) follows the DCT. Costly division is avoided by implementing quantization as a simple multiply and shift operation. Total processing required for luma encode and decode using DCT + Q + IDCT + IQ is approximately 594 additions, 16 multiplies, and 288 muxes (decisions).

The wide 128-bit bus to the ISEF can load a row of eight 16-bit prediction data in a single cycle. These instructions are performed as a single ISEF instruction where the optimizing compiler has exploited the inherent parallelism in the function.

The total number of cycles to perform these operations on a 4x4 block using a standard processor is over 1000 cycles. The same processing can be done in a software-configurable processor in 105 cycles, offering more than a 10X acceleration. In application terms, this means that a 720 x 480 @ 30 fps video stream will only require 14.2% of the software-configurable processor. Note that additional acceleration is possible by operating on larger sub-block sizes to reduce overall cycle count through increased parallelism. As an example, operating on two 4x4 blocks in parallel reduces execution time in half, dropping to 7.1% utilization.

Accelerating de-blocking requires developers to minimize conditional code. Instead of consuming cycles to determine which values to calculate — this would entail selecting which data to send to the ISEF and then creating an instruction to perform the appropriate operation — it is actually more efficient to create a single ISEF instruction that calculates all the results in hardware and then select the appropriate result. Re-ordering the 128-bit result from the IDCT stage simplifies packing of sixteen 8-bit Edge Data Pixels into a single 128-bit wide to feed the de-blocking custom instruction. Pre-calculating macro block parameters — including bS, a, B, tco and chromaEdgeFlag — is another effective optimization option supported by the state registers inside the ISEF and the instruction.

The filter’s inner loop loads the 128-bit register and executes the deblockFilter() custom instruction, computing two edges per instruction. Because the same custom instruction can be used for both horizontal and vertical filtering, there is zero overhead. This inner loop takes three cycles and is executed twice (horizontal and vertical), with about 20 cycles for loop overhead. With 64 edges in each MB of data, there are approximately 416 (64 / 4 * 26) cycles required per MB. For a 720 x 480 @ 30 fps stream, this results in 16.8 Mcycles/sec, or approximately 5.2% processor utilization.

**Motion estimation**

Motion estimation is known to consume most of the CPU budget (50-60%), but it is also very dataflow intensive. The key computation requirements are the repeated Sum of Absolute Differences (SAD) calculations used in determining the best mo-
multiplier units from the data cache. With struggle to feed the fixed arithmetic and also these CPU and DSP implementations struggle to the traditional CPU and DSP architectures. These large data sets do not many of the intermediate results needing to be reused. These large data sets do not fit well within the limited register space of the command processor and multiplier units from the data cache. With the Stretch software-configurable processor, the ISEF is capable of performing computations in parallel and holding the intermediate results in the state registers while executing fully pipelined SAD instructions.

Motion estimation consists of potentially 41 SAD and 41 motion vectors (MV) calculations per macro block. A full motion search on a single macro block requires 292K operations for D1 at 30 fps, for a total of 10.6 GOPS. By using heuristic algorithms, which is the "secret sauce" for many implementations, the developer can minimize the computations to meet target image quality and/or bit rate requirements.

Custom algorithms optimized to perform estimates across different search areas, numbers of frames, or the number of motion vectors needing to be calculated can easily be converted to ISEF instructions. A single custom ISEF instruction can replace multiple computations, as well as pipeline many of the computations using intermediate results. For example, a single custom instruction can perform 64 SAD calculations. However, the 64 partial sums could be maintained in the ISEF to reduce the number of data transfers and reuse the results in the next instruction. By pipelining ISEF instructions the compute capacity increases.

Motion estimation also involves _- and _-pixel predictions that require 9 SADs computed in 9 directions around the pixel. Although a standard formula, it is computationally intense as well. By using custom instructions, a 16x16 SAD calculation with quarter pixel precision takes 133 cycles, and a 4x4 SAD calculation with quarter pixel precision takes 50 cycles.

H.264 contains a rich set of tools for efficient video compression, making it adaptable to multiple applications and markets. However, it is computationally intense and requires an architecture that enables developers to utilize hardware acceleration. Software-configurable processors abstract hardware to the degree that developers can write application and algorithmic code in C and a 4x4 SAD calculation with quarter pixel precision takes 50 cycles.

Scalability

Because the custom instructions are created by the compiler based on C code written by developers, adapting and updating an algorithm is a straightforward process that avoids the extensive rewriting and performance profiling required when code is hand-written in assembly. As a consequence, applications developed on software-configurable architectures scale easily. For example, to convert a hand-coded assembly implementation of H.264 to a different screen resolution requires re-engineering of the algorithm. With a software-configurable architecture, the compiler does all of the heavy lifting. This flexibility is critical for rapidly evolving algorithms such as H.264. If changing an algorithm requires too much re-engineering, it is a barrier to algorithm development. A flexible architecture frees developers to experiment and discover, as well as cost-effectively implement, innovation.

Frank Lee, is a Member of Technical Staff at Stretch.