Development and optimization of high level dataflow programs: the HEVC decoder design case
Khaled Jerbi, Daniele Renzi, Damien De Saint-Jorre, Hervé Yviquel, Mickaël Raulet, Claudio Alberti, Marco Mattavelli

To cite this version:
Khaled Jerbi, Daniele Renzi, Damien De Saint-Jorre, Hervé Yviquel, Mickaël Raulet, et al.. Development and optimization of high level dataflow programs: the HEVC decoder design case. 48th Asilomar Conference on Signals, Systems and Computers, Nov 2014, Pacific Grove, United States. <hal-01120927>

HAL Id: hal-01120927
https://hal.archives-ouvertes.fr/hal-01120927
Submitted on 26 Feb 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Development and optimization of high level dataflow programs: the HEVC decoder design case

Khaled Jerbi*, Daniele Renzi†, Damien De Saint Jorre‡, Hervé Yviquel*,
Mickaël Raulet*, Claudio Alberti†, Marco Mattavelli†

*IETR, INSA Rennes, CNRS UMR 6164, UEB
20, Av. des Buttes de Coësmes, 35708 Rennes
Email: firstName.name@insa-rennes.fr
†Microelectronic Systems Lab, EPFL CH-1015
Lausanne, Switzerland
Email: firstName.name@epfl.ch

Abstract—With the standardization of the new High Efficiency Video Coding (HEVC) compression algorithm, a dataflow specification of the HEVC decoding process is also available as part of the standard. This paper presents methodologies to improve and optimize the performance of implementations derived by the dataflow specification. Regarding the architectural aspect of dataflow network, the throughput has been increased by developing more potential parallelism. For the platform aspect, critical processes have been optimized by applying SIMD functions and communications have been improved by cache efficient FIFO implementation.

Results revealed an average acceleration factor of 7 in the decoding framerate over the reference dataflow implementation.

I. INTRODUCTION

The availability of high resolution screens supporting 4K and 8K Ultra High Definition TV formats, has raised the requirements for better performing video compression algorithms. With this objective MPEG and ITU have recently finalized the development of the new High Efficiency Video Coding (HEVC) video compression standard [1] successfully addressing these demands in terms of higher compression and increased potential parallelism when compared to previous standards. So as to guarantee real-time processing for such extremely high data rates, exploiting the parallel capabilities of recent many/multi-core processing platforms is in most of the cases an obliged implementation option for both encoders and decoders. In this context, dataflow programming is a particularly attractive approach because its intrinsic properties provide the portability of the potential parallelism on different processing platforms.

The MPEG-RVC framework [2] is an ISO/IEC standard conceived to address these needs. It is essentially constituted by the RVC-CAL actor dataflow language [3] and a network language, and aims at replacing the traditional monolithic standard specification of video codecs with a dataflow specification that better satisfies the implementation challenges. The library of actors is written in RVC-CAL and provides the components that are configured using the network language to build a dataflow program implementing an MPEG decoder.

The main contributions of this work are: a) the development of an RVC-based dataflow program implementing the HEVC decoder; b) the optimization of the dataflow by exposing an higher level of potential parallelism; c) the optimization of the program for the mapping on x86 architectures using SIMD functions and efficient FIFO cache implementations. The paper is organized as follows: in Section II, an overview on the RVC framework and the dataflow HEVC decoder developed according to the RVC formalism is presented. Section III, details the methodologies used to profile and to improve the performance of the decoder. Finally, Section IV shows the implementation results on multi-core software platform.

II. BACKGROUND

The emergence of massively parallel architectures, along with the need for modularity in software design, has revived the interest in dataflow programming. Indeed, designing processing systems using a dataflow approach presents several advantages when dealing with complex algorithms and targetting parallel and possibly heterogeneous platforms.

A. Reconfigurable Video Coding

The MPEG-RVC framework is an ISO/IEC standard aiming at replacing the monolithic representations of video codecs by a library of components. The framework allows the development of video coding tools, among other applications, in a modular and reusable fashion by using a dataflow programming approach. RVC presents a modular library of elementary components (actors). An RVC-based design is a dataflow directed graph with actors as vertices and unidirectional FIFO channels as edges. An example of a graph is shown in Figure 1. Every directed graph executes an algorithm on sequences of tokens read from the input ports and produces sequences of tokens in the output ports.

Actually, defining several implementations of video processing algorithms using elementary components is very easy and fast with RVC since every actor is completely independent....
Fig. 1: A dataflow network of five processes, the vertices named from A to E, that communicate through a set of communication channels, represented by the directed edges.

from the rest of the other actors of the network. Every actor has its own scheduler, variables and behavior. The only way of communication of an actor with the rest of the network are its input ports connected to the FIFO channels to check the presence of tokens. Then, an internal scheduler enables or not the execution of elementary functions called actions depending on their corresponding firing rules. Thus, RVC ensures concurrency, modularity, reuse, scalable parallelism and encapsulation. To manage all the presented concepts of the standard, RVC presents a framework based on the use of a subset of the CAL actor language called RVC-CAL that describes the behavior of the actors.

The RVC framework is supported by a set of tools such as the Open RVC-CAL Compiler (Orcc). Orcc1 [4] is an open-source toolkit dedicated to the development of RVC applications. Orcc is a complete Eclipse-based IDE that embeds two editors for both actor and network programming, a functional simulator and a dedicated multi-target compiler. The compiler is able to translate the RVC-based description of an application into an equivalent description in both hardware [5], [6] and software languages [7] for various platforms (FPGA, GPP, DSP, etc). A specific compiler back-end has been written to tackle each configuration case such as presented in Figure 2.

B. Dataflow-based HEVC decoder

HEVC is the last born video coding standard, developed jointly by ISO and ITU, as a successor to AVC / H.264. HEVC is improving the data compression rate, as well as the image quality, in order to handle modern video constraints such as the high image resolutions 4K and 8K [1]. Another key feature of this new video coding standard is its capability for parallel processing that offers scalable performance on the trendy parallel architectures.

Such parallel capabilities offer a great opportunity to show the merits of the RVC approach. Consequently, the RVC working group has developed, in parallel with the standardization process, an implementation of the HEVC decoder using the RVC framework, which is presented in Figure 3. The description is decomposed in 4 main parts:

1) the parser: it extracts values needed by the next processing from the compressed data stream so called bitstream. The stream is decompressed with entropy decoding techniques, then the syntax elements composing the stream are extracted in order to be transmitted to the actors that they may concern. The parser applies a Context-adaptive binary arithmetic coding (CABAC) to extract the syntax element of the bitstream.

2) the residual: it decodes the error resulting of the image prediction using Inverse integer Transform (IT), which is no other than an integer implementation of the well-known IDCT [8]. The transform allows spatial redundancy reduction within the encoded residual image. As presented in figure 3, the IT can be applied on different blocks sizes (4x4 .. 32x32) and the dataflow description allows parallelizing the processes.

3) the prediction part: it performs the intra and inter prediction. Intra prediction is done with neighbouring blocks in the same picture (spatial prediction) whereas inter prediction is performed as a motion compensation with other pictures (temporal prediction). The inter prediction also implies the use of a buffer, known as Decoding Picture Buffer (DPB), containing decoding pictures, needed to perform the temporal prediction.

4) the filter: it is used to reduce the impact of the prediction

---

1 Orcc is available at http://orcc.sf.net
on the image rendering. This part contains two different filters. On the one hand, the DeBlocking Filter (DBF) [9] is used to smooth the sharp edges between the macroblocks. On the other hand, the Sample Adaptive Offset filter (SAO) [10] is used to better restore the original signal using an offset look up table.

III. Optimization of the RVC-based HEVC decoder

In order to assess the performance of the dataflow HEVC decoder presented above, Orcc has been used to generate a C implementation. The generated project is compiled with GCC and executed on a Xeon CPU at 3.2 Ghz. The preliminary results on HD streams showed a low throughput of 6.1 Frames/second. The mapping of the actors on multi-core for parallel execution did not bring scalable results.

A. Profiling

In order to better understand the bottlenecks of the design, profiling tools have been used to evaluate the workload of each actor and the obtained results have been reported in Figure 4. Results show that only 3 actors (Inter Prediction, DPB and SAO filter) consume 60% of the whole workload which means that these actors require an optimization stage and a refactoring to expose an higher potential of parallelism. In the following, an architecture optimization based on the split of the decoding process into luminance component (Y) and chrominance components (U and V) separately is presented.

B. Architecture optimizations

In the first version of the decoder, sequential decoding of the image luminance and chrominance components was applied. This description is changed to split the processes into independent actors for each image component as illustrated in Figure 5. The application of this transformation had a direct impact on the workload as shown in Figure 6 where most of the critical actors workloads became close to the rest of the design, such as 6% for the DPB-Y and 11% for SAO-Y. Concerning the Luminance component of the Inter Prediction,
application. In this work, we used Intel SIMD instructions to
target x86 architectures. Our techniques relies on an annotation
mechanism in order to keep the portability of the high-level
description over multiple platforms:

1) First, the developer identifies the code to optimize and
move it in its own procedure, knowing that the optimized
kernels have to use the same parameters than their
equivalents in CAL. The optimized version should be
available into an external library (such as FFMPEG).
2) Then, the developer adds the directive @optimize on
top of the CAL procedure to identify the optimized
version of the procedure (see Listing 1). The directive
is based on the following syntax @optimize(condition=
"CONDITION", name="NAME") where NAME is the name of
the optimized kernel and CONDITION is a predefined
condition that enable the execution of the kernel.
3) Finally, the generated code can use the optimized kernels
when they are available (see Listing 2).

```
@optimize(condition="defined(OPENHEVC_ENABLE)", name="put_hevc_qpel_h")
procedure put_hevc_qpel_h_cal(int size=16) arg1[64*64], int arg22
begin
  // Kernel body in CAL
  (...)end

Listing 1: CAL code
```

```
void put_hevc_qpel_h_cal(i16 arg1[4096], i32 arg2) {
#if defined(OPENHEVC_ENABLE)
  put_hevc_qpel_h(arg1, arg2);
#else
  // Standard kernel
  (...)endif

Listing 2: Generated code
```

As a result, optimized applications easily stay compatible with
all backends and platforms.

To link with SIMD functions, the CAL code undergoes
small modifications by adding annotations and by correspond-
ing functions they become identical to SIMD ones (arguments
number and types). As explained in Figure 7, the FFmpeg
SIMD functions are compiled to obtain a dynamic library.
Then using pretty printing, a correct link of the C project
with the dynamic library is guaranteed.

2) Cache-efficient FIFO channels: In software, FIFO chan-
nels are traditionally implemented by a circular buffer allo-
cated in shared memory. Read and write are then achieved
by accessing the buffer according to read and write indexes
that are updated afterwards. The state of FIFO channels is
known by comparison of their indexes. Using circular buffer
to implement FIFO channels avoids side shuffles of data
after each reading, but implies an advanced management of
memory indexes that may ultimately lead to poor performance.
In modern general-purpose processors, the processor cores
usually communicates through common shared memory ac-

![Diagram](chart.png)

Fig. 7: Conception flow of the SIMD linked implementation
shows a compilation of a RVC-CAL code with annotations
linked with a DLL library generated from the compilation of
FFmpeg.

IV. RESULTS

To apply the optimization methodologies evoked above, an
Intel Xeon CPU at 3.2 GHz with 6 cores has been used. We
applied the HD stream “Kimono” with Low Delay encode
and QP=27. The HD stream “Kimono” with Low Delay
encode and QP=27 were the test sequence selected for the
experiments. In Table I, all the results of the reference design
and the YUV design with and without x86 optimizations are
presented.

<table>
<thead>
<tr>
<th>Description</th>
<th>Padding</th>
<th>SIMD</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>OFF</td>
<td>OFF</td>
<td>6.1</td>
<td>8.4</td>
<td>8.5</td>
<td>8.2</td>
<td>9.4</td>
<td>9.5</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>OFF</td>
<td>6.1</td>
<td>9</td>
<td>9</td>
<td>8.8</td>
<td>10.6</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>ON</td>
<td>8.3</td>
<td>11.9</td>
<td>9.9</td>
<td>8.1</td>
<td>7.9</td>
<td>8.5</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>ON</td>
<td>8.3</td>
<td>13.1</td>
<td>14.8</td>
<td>15.6</td>
<td>15.3</td>
<td>15.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YUV</th>
<th>OFF</th>
<th>OFF</th>
<th>9.5</th>
<th>16.7</th>
<th>19</th>
<th>22.6</th>
<th>22.7</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ON</td>
<td>OFF</td>
<td>9.5</td>
<td>17</td>
<td>23.6</td>
<td>23.2</td>
<td>23.4</td>
<td>24.1</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>ON</td>
<td>15.8</td>
<td>27.7</td>
<td>37.4</td>
<td>30.4</td>
<td>35.9</td>
<td>35.5</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>ON</td>
<td>15.8</td>
<td>28.8</td>
<td>40.5</td>
<td>43.2</td>
<td>46.6</td>
<td>39.1</td>
</tr>
</tbody>
</table>

To focus on the evolution of the framerate, the curves
of Figure 9 are considered. The impact of the YUV design is
the scalability with the number of cores. The framerate
reaches a maximum of 46.6 FPS with 5 cores then starts
decreasing and this is due to the increasing communication
cost between processing cores. To compare the obtained result
with a reference from the literature, the same stream on the
OpenHEVC decoder [12] has been applied. Using one thread,
OpenHEVC decodes at 59 FPS in mono-thread which is close
Fig. 8: A smart FIFO implementation in the cache of the processor. In (a), the classic definition of the indexes results in setting indexes in the same line of cache which requires useless waits for refresh. In (b), by adding the paddings, the indexes are set into distinct lines of cache.

Table II: Decoding acceleration of the reference and the YUV designs on multi-core processor (in FPS) using all combinations of enabling and disabling SIMD functions and cache padding.

<table>
<thead>
<tr>
<th>Description</th>
<th>Padding</th>
<th>SIMD</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>ON</td>
<td>OFF</td>
<td>1</td>
<td>1.47</td>
<td>1.47</td>
<td>1.44</td>
<td>1.73</td>
<td>1.73</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>ON</td>
<td>1.36</td>
<td>1.95</td>
<td>1.62</td>
<td>1.32</td>
<td>1.29</td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>ON</td>
<td>1.36</td>
<td>2.14</td>
<td>2.42</td>
<td>2.55</td>
<td>2.50</td>
<td>2.47</td>
</tr>
<tr>
<td>YUV</td>
<td>OFF</td>
<td>OFF</td>
<td>1.55</td>
<td>2.73</td>
<td>3.11</td>
<td>3.70</td>
<td>3.72</td>
<td>3.11</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>ON</td>
<td>1.55</td>
<td>2.78</td>
<td>3.86</td>
<td>3.80</td>
<td>3.83</td>
<td>3.95</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>ON</td>
<td>2.59</td>
<td>4.54</td>
<td>6.13</td>
<td>4.98</td>
<td>5.88</td>
<td>5.81</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>ON</td>
<td>2.59</td>
<td>4.72</td>
<td>6.63</td>
<td>7.08</td>
<td>7.63</td>
<td>6.40</td>
</tr>
</tbody>
</table>

Fig. 9: Decoding framerate using the different optimization techniques compared with the reference description.

Table: Decoding acceleration of the reference and the YUV designs on multi-core processor (in FPS) using all combinations of enabling and disabling SIMD functions and cache padding.

<table>
<thead>
<tr>
<th>Number of processor cores</th>
<th>Description</th>
<th>Padding</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reference</td>
<td>ON</td>
<td>OFF</td>
</tr>
<tr>
<td>1</td>
<td>YUV</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>2</td>
<td>YUV+SSE</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>3</td>
<td>YUV+cache</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>4</td>
<td>YUV+SSE+cache</td>
<td>OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

REFERENCES