Acceleration of Reconfigurable Video Coding Using New Parallel Architectures

Maria Pantoja and Nam Ling
Department of Computer Engineering
Santa Clara University
Santa Clara, CA, USA
Email: {mpantoja@scu.edu, nling@scu.edu}

Abstract- In this paper we present a novel technique to accelerate reconfigurable video coding with parallel architectures. We focus on the use of the Graphics Processing Unit (GPU) as our platform for parallel processing but the algorithm can be implemented on other parallel architectures. Implementation of the solution shows that execution time is reduced 16-60% depending on the decoder module implemented on the GPU.

Index Terms- Transcoding, reconfigurable video coding, data flow, video coding, GPU, CUDA

I. INTRODUCTION

The different video coding standards introduced in recent years have contributed to the tremendous increase in applications requiring multimedia support. The MPEG Reconfigurable Video Coding (RVC) [1] is an initiative started in 2004 by the MPEG committee whose goal is to provide first a framework that will allow a faster and more flexible path to the innovation of the MPEG standards; and second, a high level specification model for software and hardware synthesis. The current specification of MPEG codec is inadequate for new design challenges and fast implementations of improvements to the standards. Parallel multicore computers are ubiquitous in the market but video standards have not developed models to take advantage of these new architectures, leaving the hardware under exploited by sequential and non-reconfigurable designs. MPEG RVC has developed two standards [2, 3]. Firstly, the Video Tool Library [2] specifies a set of Functional Units (FUs) commonly used on video decoders, such as transforms, motion compensation and entropy coders. The FUs are described using a data flow oriented language CAL [4] and a language similar to XML for the description of the FUs’ configurations used to create a decoder. Secondly, the codec configuration representation [3] defines the Decoder Description Language (DDL), and the format of the coded bitstream -- Bitstream Syntax Description Language (BSDL). The basic idea of how RVC works can be seen in Figure 1. For the RVC decoder to work we need to send the input bitstream together with the BSDL and DDL specifications, and then use the Video Tool library to generate the output bitstream. Thanks to this structure RVC allows, new FUs can be used without additional standardization steps.

There are several articles using RVC to solve different video coding problems. An excellent overview of RVC can be found in [5]. Article [6] presents an in-depth case study on dataflow-based analysis and the exploitation of parallelism in the design and implementation of an MPEG RVC decoder; the paper also describes a software tool (developed by the authors) for analyzing dataflows networks targeted for multicore platforms. In [7], the authors propose an efficient hardware-assisted syntax-decoding model for software based video decoder. The proposed model is a generic model for different video codec standards although it was implemented specifically for H.264. In [8] a synthesis tool that transforms a CAL specification into C code is implemented. Article [9] describes possible decoder reconfigurations within the RVC framework to satisfy a wide variety of different applications. In [10], it presents a verification technique that minimizes the number of test patterns but at the same time covering multiple profiles based on the functional commonalities extracted from multiple coding standards. In [11] an alternative framework to the MPEG RVC called dynamically configurable video compression is developed, the coding tools themselves may be created, configured and re-configured adaptively.

In this paper we present a novel technique to accelerate the execution of RVC by using new parallel architectures; in particular we use the GPU for proof of concept, which can also be translated to other similar parallel architectures. The latest GPUs are highly parallel, multithreaded, manycore processors with tremendous computational power and high memory bandwidth. The advantage of using a GPU is that they are designed to have parallel “manycore” architecture; each core is capable of running thousands of threads simultaneously. In an application like RVC where exploiting parallelism is one of the main objectives of the standard, GPU computing seems like the perfect choice.
The paper is organized as follows. Section II presents a brief overview of RVC and Compute Unified Device Architecture (CUDA) parallel computing engine. Section III describes the method proposed in this paper. Results and conclusions are presented in Sections IV and V respectively.

II. OVERVIEW OF CUDA AND RVC

A. GPU Computing Using CUDA

In order to better exploit the GPU device, compute unified device architecture (CUDA) [12] was developed by NVIDIA as a parallel computing architecture for their GPU. CUDA is accessible to developers through standard programming languages C/C++. Three key abstractions that must be exposed by the programmer are:

1. Hierarchy of threads groups.
2. Shared memories.
3. Barrier synchronization.

CUDA allows the programmer to define C functions, called kernels, that, when called, can be executed directly on the GPU in parallel using \( N \) threads. A new kernel is defined using global declaration and the number of CUDA threads for each call is specified using the \( \text{<<<N, M, parameters>>=} \) syntax. Limited memory resources of a processor core (512 on current GPUs) restrict the maximum number of threads per block. Each thread has a private local memory; each thread block has a shared memory visibly to all threads in the block. All threads have access to same global memory.

B. Reconfigurable Video Coding (RVC)

RVC provides a framework to allow content providers to define a multitude of different codecs by combining blocks (FUs) from a standard library. A description of the decoder is associated to the encoded data, enabling reconfiguration and instantiation of the appropriate decoder.

In this paper we are particularly interested in FUs created using the CAL description language, and we evaluate the benefits of having these libraries implemented in CUDA. CAL is a dataflow and actor oriented language. An actor is a modular component that encapsulates its own state. Interactions between actors are only allowed through input and output ports. The behavior of an actor is defined in terms of a set of actions (consume tokens, modify internal state, produce tokens). An actor performs computation as a sequence of atomic steps called firings. Basic structure of a FU network (group of actors) as a block diagram can be found in Figure 2. Its textual description can be found in Figure 3. A basic FU CAL actor is shown in Figure 4.

III. ACCELERATING RVC USING CUDA

In this paper we accelerate RVC using CUDA, and as a proof of the values to creating a library of FUs using CUDA we implement a few FUs written in CAL in CUDA, exposing the parallel programming “friendliness” of the CAL language. The basic FUs that we choose to implement are clip and transpose. A clip clamps the input values of a 16×16 matrix to \([0, 255]\) or \([-128, 127]\) if signed. Transpose calculates the transpose of a 16×16 input matrix. The implementation of all FUs needed for an H.264 decoder is beyond the scope of the paper.

![Fig. 1 MPEG RVC decoding.](image1)

![Fig. 2 FU network block diagram - each block is an actor/FU created using CAL.](image2)

![Fig. 3 FU network description using RVC FNL.](image3)
For each of the actions in the FU we create a CUDA kernel; and for each of the 256 (16 × 16) input values we create a CUDA thread. To avoid having to synchronize all CUDA threads we use the priority function that will also execute as a kernel on the GPU. The transformation from CAL to CUDA for FUs with high data parallelism, which is the case for clip and transpose, is relatively straightforward. This data is implicitly parallel and can easily exploit a single instruction multiple data (SIMD) implementation. As explained in [14], programmers must find efficient mappings of their applications to a CUDA thread-based model, even if the data show a high degree of data parallelism; the programmer also has to make an efficient use of the GPU memory hierarchy, and this part is not directly accomplished from the CAL FU description, undermining to a little extent the usability of implementing an RVC on CAL. For example, FUs with data dependencies, like those of the inverse discrete cosine transform (IDCT) of size 8×8, we have to expose memory hierarchy structures to really get the best performance out of the GPU, not a simple task to get from the CAL description language.

IV. EXPERIMENTAL RESULTS

To run the experiments we used an HP Pavilion dv7 3000 with an Intel CPU core i7 (4 cores, processor base frequency 1.6 GHz, and 64-bit instruction set) and an Nvidia CUDA enable GPU GeForce GT 230M (48 cores, 158 GigaFlops, and processor clock of 1100 MHz). CUDA profiler tool was used to analyze the execution of the FUs.

The modules computed clip and transpose have been implemented in custom “.cu” CUDA files. The DCT 8×8 module was part of NVIDIA SDK examples and was used as a comparison to see how a more complex FU would execute in CUDA. The average GPU usage was 16% for the clip FU, 35% for transpose and 58% for the dct8×8. Table I shows the CPU and GPU usage in microseconds. It can be seen that we obtained better performance with increased complexity of the module, and by using a parallel architecture including modules with high data dependencies as the DCT module. The third column in the table shows the memory usage per FUs.

Figure 5 presents a diagram for all subroutines called in FUs’ clip; the diagram was obtained using CUDA profiler and is useful in analyzing which of the subroutines take more CPU/GPU time. In the case of the clip FU is the priority function since it has to wait for the rest of the kernels to write its output.

V. CONCLUSION AND FUTURE WORK

We present, in this paper, an initial step to implement RVC on a highly parallel processor using the CUDA technology. We found that using CUDA increases the processing speed of the FUs studied here by 16 to 60 times. There is a clear advantage in exposing parallelism in the decoder description software, and CAL works well in exposing this parallelism, although it is not a trivial operation to translate from CAL to C or to a hardware synthesis language. The work in this paper and those in the references initiated such implementation of RVC, although it would take time to see commercial implementation of products out of this.

There are some works already published that translate CAL to C, so it is also an interesting future work to create a CAL extension that accepts CUDA kernels, CUDA kernels will be automatically extracted from the RVC description. We also plan to implement the H.264 decoder entirely on CUDA so we can perform peak-signal-to-noise ratio (PSNR) analysis.

Table I

<table>
<thead>
<tr>
<th>FU name</th>
<th>CPU time in µsec</th>
<th>GPU time in µsec</th>
<th>Global memory read throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clip</td>
<td>48.99</td>
<td>176</td>
<td>0.151</td>
</tr>
<tr>
<td>Transpose</td>
<td>350.23</td>
<td>4500</td>
<td>9.232</td>
</tr>
<tr>
<td>DCT 8×8</td>
<td>28660.48</td>
<td>35714</td>
<td>23.135</td>
</tr>
</tbody>
</table>
Fig. 5 GPU time per function call for the Clip FU.

REFERENCES


