Function Similarity Evaluation Tool

CSEE E6861y - Computer-Aided Design of Digital Systems

Kshitij Bhardwaj
kb2673@columbia.edu

Paolo Mantovani
pm2613@columbia.edu

ABSTRACT

This tool evaluates the similarity of two functions, which have the same support, given two covers representing the functions. It

The main purpose of this project is to evaluate the efficiency and performance improvements that can be obtained by integrating hardware accelerators within a system framework, based on a tiled Network on Chip (NoC) and designed using High Level Synthesis (HLS) tools. In particular, we show that a hardware accelerator, synthesized from SystemC and integrated into the SoC, can outperform a processor, with comparable parallelism, executing the same algorithm in software. We report experimental results for both level of abstraction: System level and RT level.

Keywords

SoC Design, Hardware Accelerator, High Level Synthesis

1. INTRODUCTION

It is well known that hardware accelerators are widely integrated in modern embedded systems. The classic RTL design flow, though, does not seem to be suitable any more for a fast design of a complex and heterogeneous system. In this project we exploit the case study of an image processing algorithm (the *JPEG decoder*) to evaluate a complete design flow, which begins form a specification, written by a third party in a high level language, and finishes with a synthesizable RTL hardware implementation of the same algorithm. Intermediate steps allow to validate the design at different level of abstraction and to estimate its performance within a complete system framework.

The structure of the report follows the phases of the project. In section 2 we start by analyzing the system framework provided. Section 3 introduces the JPEG decoder and the CHStone [**?**] benchmark suite. Then we briefly describe the porting from plain C to SystemC and the accelerator structure. At the end of the section we report the results obtained by running Cadence HLS tool *CtoS*. In section 4 we report the experimental results and finally in section 5 we conclude with the lessons learned and with strengths and weaknesses of this design flow.

2. SOC FRAMEWORK FOR *PLUG & PLAY*

The starting point of the project is a tiled based NoC designed in SystemC. It can host multi-core processor tiles, composed by 4 "MIPS like" cores with level 1 and level 2 caches, described at the RTL, and hardware accelerator tiles, with dedicated "scratch pad" memory. Each tile has also a Network Interface (NI), which provides a uniform protocol to connect third party accelerators, assuming that they are wrapped within a Transaction Level Modeling (TLM) module, with TLM FIFO for data I/O transfers. The NI handles the communication through the NoC by implementing software primitives for explicit message passing communication among the tiles. The processor tiles have therefore the capability to delegate the available accelerators to complete certain tasks. The details about the network protocol and the implementation of the tiles are described in [**?**].

3. CHSTONE JPEG DECODER

The algorithm chosen to be accelerated is the JPEG decoder from the CHStone [**?**] benchmark suite. The choice of the CHStone is motivated mainly by the fact that these benchmarks are explicitly meant to test the efficiency of HLS tools. Also, the source code is plain C and it includes the test vectors, therefore it can be easily compiled for the MIPS like core available within the NoC, which doesn't support an OS. Among the algorithms of the suite, the JPEG was the best candidate, because of its complexity. Considering, in fact, the communication overhead deriving from the NoC and the NI, we must assume that an accelerator will be more efficient, in terms of performance vs. power, only if the computational complexity of the task is large enough. For small and simple algorithms, instead, a pure software implementation, which does not require data transfer through the NoC is expected to be more efficient. Moreover we derive the TLM SystemC accelerator from the C code, because existing implementations of SystemC accelerators are usually not suitable for being executed in software, without strongly modifying the kernel of the algorithm. Our purpose, in fact, is not to build a state of the art accelerator, but to prove that through this new design flow we can increase productivity and performance, guaranteeing at the same time a significant power saving.

3.1 Software implementation

We shortly describe CHStone JPEG decoder structure. The first function call parses the information contained in the JPEG header, which has undefined size. It is therefore compulsory to load all the coefficients and to build a few tables before starting the decoding process. Special characters, called markers, identify the beginning of different sections, which provide basic information about the picture (such as the size), the coefficient used for compression and quanti-

zation, quality parameters and finally the compressed data. The other functions implement the baseline JPEG decoder and use the information provided in the header to reconstruct the original RGB table (notice that the compression is "lossy"). The complete JPEG standard is included in [**?**].

A small test vector was statically included within the CHStone code. In order to obtain significant experimental results, however, we wrote a small preprocessor that parses different and bigger images and outputs a configuration header for the test bench used for simulations.

3.2 SystemC implementation

In this section we list the major changes applied to the CHStone code, during the porting process to synthesizable SystemC. Fig. **??** is a block diagram representation of the SystemC code, intended to help in understanding the interaction between the processes and how the accelerator is interfaced to the standard TLM FIFO for being plugged to the NoC. Porting to SystemC included the following steps:

- Split of the sequential C function into smaller concurrent behaviors, which are synchronized through *sc_signals* and a handshake protocol. Data transfers are enabled by shared local memories which act as temporary scratch pads.
- Conversion of global variables into shared attributes of the module, which are accessible by every process declared within the System-C module.
- Replacement of all the pointers with array indexes declared as member variables. This allows the HLS tool to parse and build the code correctly.
- Enabling the overlap of I/O transfers and computation. This allows to decode bigger images, because the picture does not need to be stored entirely within th accelerator.
- Generation of the output file header.
- Implementation of the TLM wrapper to connect the accelerator to its test bench. The TLM interface is also required to integrate the accelerator into the NoC.
- Implementation of a configurable test bench, which can feed the accelerator with different input images and checks the output, with respect to a golden output provided.
- Optimization of loops to enable further options for design space exploration with the HLS tool.

	Memory	**Logic**	**TLM**	**Total**
area mm^2	0.9786	0.1139	0.0099	1.1020
power mW	47.495	274.99	1.3222	323.81

Table 1: High Level Synthesis of the JPEG decoder, constrained with a 1GHz clock frequency

3.3 From SystemC to RTL

After coding and testing the accelerator with the fast SystemC simulations, we used our code as input for CtoS, which is able to generate RTL verilog. CtoS implementation was set to be "target aware", thus the tool is allowed to add resources taken from the standard cell library and states (i.e. it increases the latency), in order to meet the timing constraints. Since we aim to integrate the accelerator into the existing SoC, the given constraints are the technology (*32 nm PD-SOI*) and the target clock frequency (1 GHz).

The most difficult aspect in configuring the HLS is dealing with arrays which need to be mapped as static ram blocks, when their size is not feasible for registers. All the buffers, drawn in light blue in Fig. **??** have been mapped to vendor static ram memories, whose liberty file and verilog view are provided by a memory generator for the same technology used for synthesis.

The need of a significant number of memories, reduces the possibility for design space exploration, because the HLS tool is very conservative while scheduling memory accesses. Also memory ports are limited and the buffer chain represented in Fig. **??** was carefully designed to avoid conflicts between processes for memory accesses.

After scheduling we ran a more precise synthesis tool on the generated RTL and we obtained area and performance reported in Table 1. The most relevant information provided is that almost 89% of the area is memory. This is actually common for hardware accelerators, because they generally target a single application which operates on large amount of data.

4. EVALUATION

In this section we first focus on the high-level simulation of the accelerator, integrated in a modified version of the Rabbits Virtual Platform (VP) framework [**?**, **?**]. The platform models the SoC described in Sec. 2, even if the available processor model for Rabbits is a state of the art ARMv6 core, instead of the "MIPS R2000" provided at the RTL level. Actually for our case study the two processors reported a similar IPC, probably because of the small instruction level parallelism offered by the algorithm. Also, we are aware of the fact that RTL simulation is cycle accurate, whereas VP is not. Nevertheless simulating on Rabbits allowed us to compare the process of decoding big images, taking into account the overhead of a real system (i.e. OS, device driver, communication, memory latency). Such a simulation, when the picture size grows, is not feasible at the RTL level, because the running time explodes.

We then report data concerning the RTL simulations.

Table 2 summarizes simulation results with respect to the JPEG weight, the image size and the compression ratio.

4.1 System Level

Firstly we try to obtain a baseline as a comparison for our JPEG decoder evaluation. Therefore we compile the CHStone original C code for the VP ARM processor and we run it for different image sizes and compression ratios. The

Image	Weight	Size	Compression	RTL ACC only	VP only	VP with ACC	VP Speedup
chstone	5.1K	90x59	32.67%	327521	12799005	9707851	1.31
lena	6.6K	256x256	3.44%	2588629	67969104	24297449	2.79
fire	12.0K	340x340	3.54%	4671280	122417017	38520484	3.17
orange	27.0k	800x496	2.32%	15476731	393519047	110264509	3.56
moon	40.0k	900x900	1.69%	31481890	784515821	218384556	3.59
fish	107.0k	1152x864	3.67%	39859785	1037381360	270253629	3.84
flower	85.0k	1600x1200	1.51%	73838862	1830097863	502503637	3.64
hdcol	166.0k	1920x1080	2.73%	81696290	2086998030	550907798	3.78

Table 2: Jpeg decoder tests, using a set of images, ordered by size. We report latency, in terms of clock cycles for the RTL version, simulated through CtoS, for the CHStone software, running on the ARM modeled in the Rabbits VP, and for the accelerated version, also running on the Rabbits VP.

column *VP only* in Table 2 reports the cycles measured[1] during this simulations.

The setup for the next step is the following. In addition to the TLM interface, we include in our SystemC a few configuration registers, accessible by the NI, which will be written by a message sent from the processor tile, when the core is asking the accelerator for some service. Similarly, the core can poll another register to check whether new data are available to be retrieved and stored in its local L2 cache. Another small logic block, required to plug in the accelerator is a data width adapter. Its task is on one hand to collect data from the messages' payload and to build the 64 B data token for the JPEG decoder when feeding its input. On the other hand, this logic receives the output data token from the decoder and must prepare the message payload for the NI.

We then need to write and compile a piece of code for the ARM processor. This software routine initializes the accelerator and starts transferring data from memory to the decoder, using DMA. A second thread polls the accelerator, waiting for output data to be ready. To enhance simulation speed, the output is retrieved and then discarded, without saving the bmp file.

Fig. **??** presents the latency of the accelerated JPEG on the VP. Clearly the number of clock cycles grows linearly with the image size, which is consistent with the expected asymptotic running time of the algorithm. The offset is attributable to the initialization and communication overhead. The graph shown in Fig. **??** displays finally the speedup of the accelerated simulation with respect to the baseline described above. Although the trend seems to be logarithmic, the points deviate significantly from the curve. The reason is because compression ratio (JPEG weight) depends also on the specific information and features of the original picture. Obviously, the more details have to be encoded, the less efficient will be the algorithm. A second issue that deserves attention is the fact that when the size of the picture grows too much, while the compression ratio is very low (good compression), the overhead for moving data can dominate the overall latency.

As an anticipation to the next section, we remark that the reported speedup is a lower bound, because we do not capture all the micro-architectural optimizations included in the final RTL implementation.

[1]Rabbits is not cycle accurate

4.2 Register Transfer level

The setup for the RTL simulation should be similar to the one described in the previous section for the System level. The system and the hardware wrapper for the accelerator are still conceptually the same, but the simulation at this level is cycle accurate and running time is considerably slow. Therefore we were able to run the software CHStone on the processor tile only using the smallest image available. The co-simulation of the accelerator and the rest of the system was split into two separate simulations: using a dummy accelerator, that fetches the input and loops back to the output, we measured the communication overhead for initializing the accelerator, sending and receiving the packets; from CtoS console, we then ran the RTL simulations of the decoder with different pictures, but connecting a test bench to the TLM FIFO. Also, being the processor different, we needed to recompile the CHStone for the baseline and the software routine used with the dummy accelerator.

The graph in Fig.**??** shows the growth of the latency, with respect to the picture size. Not surprisingly, we obtain again a linear trend, but it is worth to notice that the coefficient measured for the System Level simulation is more than six times larger. This is due to two main reasons: first it confirms that the micro-architectural optimization allowed by the HLS tool is relevant to the performance gain achievable; second we must consider that the test bench is able to push inputs to the queue at the maximum sustainable rate for the accelerator, while the VP might not. We couldn't measure the communication throughput between the core and the accelerator, but we can assume that occasionally the decoder could be stalled, waiting for the next data token.

As mentioned, we could measure the running time for the software version only with the smallest picture, which is the one included in CHStone code. Thus we could calculate the accelerator speedup for this picture, which is 18.4×. Clearly the RTL implementation exploits micro-architectural optimization, but if we were able to simulate different inputs, we could better analyze the impact of the picture size and check whether the speedup still grows with a logarithmic trend. Enhancing simulation speed for the overall SoC is left as a future work. At the RT level we were also allowed to measure the energy consumed by the accelerated execution, including both processor and accelerator tiles energy, with respect to the energy of software simulation with the

processor only. The energy saving is remarkable and stands above 90%. It is also worth to mention that about two thirds of the overall energy is imputable to communication.

5. CONCLUSIONS

In this project we implemented a hardware accelerator for JPEG decoding in SystemC. We exploit this case study in order to evaluate a design flow that takes advantage of a pre-existing SoC infrastructure, also designed in SystemC and equipped with an RTL implementation of a processor tile. The design was then synthesized through a HLS tool, in order to estimate area and power consumption. Finally we measured through simulation the speedup at different level of abstraction (and therefore different accuracy), proving the effectiveness of the design flow and the potential benefits achievable with a heterogeneous SoC with hardware specific accelerators.

Despite the well promising results, a weakness of this flow consists in the slow RTL simulation, which do not allow a complete analysis of performance for large test vectors. Also the HLS tool is not enough flexible when dealing with memories and design optimization was considerably limited. Finally we should mention that many features of SystemC, which enhance simulation speed and allow a quick porting of the code from another high level language, are lost when the designer needs to target the synthesizable subset of SystemC.