Midterm Report: Design an accelerator for Computing MRI-Q Matrix

COMS E6868 - Embedded Scalable Platforms - Spring 2020

Pei Liu
pl2748@columbia.edu

ABSTRACT

I plan to design an accelerator to calculate the Magnetic Resonance Imaging Q matrix through SystemC and HLS tool Stratus and implement this design on FPGA. Then I will explore the design space to get a thorough understanding of the methodology of designing accelerators through ESP.

1. INTRODUCTION

Magnetic Resonance Imaging is commonly used by the medical community to safely and non-invasively probe the structure and function of human bodies. Images generated using MRI have a profound impact both in clinical and research fields. The reconstruction of non-Cartesian trajectory sampling data is faster and less sensitive to imaging artifacts caused by non-Cartesian trajectories than sampling in Cartesian space, but it increases computation significantly [1]. The computation for the MRI-Q matrix is an important algorithm used for image reconstruction with non-Cartesian trajectory sampling [2]. So accelerating MRI-Q matrix computation helps MRI to benefit human beings.

1.1 Motivation

Heterogeneous systems architecture is the future trend. Hardware specialization can bring order-of-magnitude more energy efficiency. Designing an accelerator for computing the MRI-Q matrix through ESP is a good way to learn the ESP design methodology.

2. SPECIFICATION

The algorithm for computing the MRI Q matrix, shown in Fig. 1. The main accelerating direction is to unroll the for-loops. The programmer view algorithm C code and input data set come from the Parboil benchmark suite [3].

2.1 Assessment

The goal of the assessment is to return correct Q matrix implemented on FPGA and compare the executing time of CPU and the accelerator designed. Also, several different designs will be implemented on FPGA to explore the design space.

2.2 Milestones

1. Analysis of the algorithm and the programmer's implementation in C (by Feb. 19)– Done!
2. Learning two tutorials: "How to design an accelerator in SystemC (Cadence Stratus HLS)" and "How to design a single-core SoC"[4]. (by Feb. 28) – Done!
3. High-Level-Synthesis implementation in SystemC (by Mar. 11)– Done!
   - Transform programmer view algorithm to HLS-ready SystemC

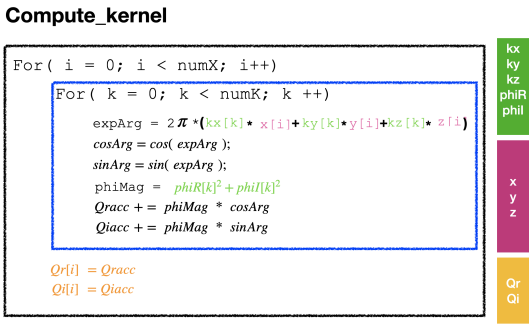

Figure 1: Algorithm for computing MRI Q matrix [1]

   - Initial HLS using the Cadence Stratus tool
4. Evaluation on an FPGA platform (by Mar. 25)
5. Mid-term presentation and report (by Mar. 25)
6. Initial Design Space Exploration (by Apr. 15)
7. Enhanced Design Space Exploration (by Apr. 22)
8. Final refinement and analysis (by May 5)
9. Final presentation ($\sim$ May 11) and report ($\sim$ May 15)

2.3 Critical Aspects

1. Implement sine and cosine functions in the innermost for-loop.
2. Try different methods of optimization to reduce latency or decrease area.

3. PROGRESS

3.1 Programmer View Implementation

The Parboil benchmark provides datasets to run the programmer view implementation, shown in Table 1. For input data with image size "small" and "medium", the execution takes 4.7s and 18.7s running on the google server (socp03), respectively. But for the "large" dataset, the run time is 186685s $\sim$ 2.2 days. The higher precision of the reconstructed image, the bigger the input data size, and the more computation.

name	image size	# of pixels	K-space dimension
small	32*32*32	32768	3072
medium	64*64*64	262144	2048
large	128*128*128	2097152	2097152

Table 1: Datasets of MRIQ from Parboil Benchmark

3.2 Behavioral Simulation

I generated skeleton code from the ESP template with two configurable registers: numX and numK, which corresponds to the # of pixels and K-space dimemsion in Table 1. Then I edited the code in the following files:

- **mriq/hw/src/fpdata.hpp**
  Add fpdata.hpp to src/ folder. Typedef FPDATA. Defined functions used to do datatype conversions, for example, int2fp(), fp2int(), fp2bv(), bv2fp(). These functions are used in both the testbench and different processes of the accelerator design. In the testbench, data in floating-point type read from files is converted to fto fixed-point representation, then converted to sc_bv to be transported through the network-on-chip. In accelerator, data in sc_bv type is converted to sc_int to be stored in PLM. In computation phase, sc_int data is converted to fixed -point representation.. After computation finished, The fixed-point data needs to be converted back to sc_int in order to be stored in PLM. Then, it is further converted to sc_bv to be transported back to the testbench for validation. In testbench, sc_bv is converted to fixed-point representation. and further converted to floating point type to be compared with the golden output data.
- **mriq/hw/src/mriq.hpp**
  I declared 10 PLMs, with name plm_x, plm_y, plm_z, plm_kx, plm_ky, plm_kz, plm_phiR, plm_phiI, plm_Qr, plm_Qi, and declared functions *mySinf*, *computeQ*, *load_one_data*, and *store_one_data*, used in mriq.cpp.
- **mriq/hw/src/mriq.cpp**
  Rewrote the processes *load_input*,*compute_kernel*, and *store_output*.
- **mriq/hw/src/mriq_functions.hpp**
  Wrote function *ComputeQ* which is the key computation part of Q matrix, showed in Fig.1. Wrote function *mySinf*, which is to compute sine. In sine function, first convert an input value to $0 \sim \pi/2$ range, then find the closest data point in the sin_table. For now, I use the interpolation method to get the sin(x). *load_one_data* function is to load one variable into one PLM. Similarly, *store_one_data* is to store one variable into one PLM.
- **mriq/hw/tb/system.hpp**
  Declared the paths for input file and golden output file. Modified the SC_HAS_PROCESS constructor. Declared parameters and functions used in system.cpp file.
- **mriq/hw/tb/system.cpp**
  Rewrote functions ***load_memory***, ***dump_memory***, and ***validate***. Added function ***inputData*** to read input data from file, and ***outputData*** to read golden output data from file.
- **mriq/hw/tb/sc_main.cpp**
  Added the paths and names of the input file and the golden output file.

Behavioral simulation succeeded when running the whole dataset (32*32*32). It takes a few hours to run since the fixed-point datatype is not native to c/c++.

3.3 RTL generation

I have generated RTL successfully by running make mriq-hls in the working folder. In this process, all the print sentences in the files in the src/ folder should be deleted to avoid that the ESP_REPORT causes errors in this process. In every for-loop, there should be a wait() sentence after the "for statement" to allow Stratus to perform loop unrolling.

3.4 RTL Simulation

By running an accelerated simulation method, I have tested the generated RTL without error. Instead of computing the whole dataset, which has 32K pixels, I edited the code to compute only the first 4 pixels in the input dataset and reduced the k-space size from 3K to 16. The simulation of RTL succeeded. My accelerating method of simulation is to edit the default value of (numX, numK) from (32768, 3072) to (4, 16) both in tb/system.hpp and src/mriq_conf_info.hpp.

4. FUTURE WORK

4.1 Accelerator Integration

Finish the accelerator integration and prototpye on FPGA.

4.2 Design Space Exploration

There are two directions for design space exploration. The first is to reduce latency. The optimization methods I plan to do is as follows:

1. Loop unroll and pipeline.
2. Constrain latency.
3. Reduce fixed-point precision.
4. Add more ports to PLM to increase parallelism.
5. Optimize compute function. Optimize the computation of sine and cosine functions.

The second direction is to reduce area. The following are the methods I plan to try.

1. Use ping pong buffers to load a portion of data to do computation. Load all the frequency-related variables (kx, ky, kz, phiR, phiZ) into PLM, but only load a portion of pixels (x,y,z).
2. In place storage of output.
3. Use different types of PLM blocks. frequency-related variables are stored in one type of PLM with size numK. The pixel variables (x,y,z) could be stored in a much smaller PLM block if using ping-pong buffers.
4. Reduce sine look-up table.
5. Reduce fixed-point storage. For now, data size is 32 bits. Later, I will try to reduce this data width while guaranteeing the computation accuracy. A smaller number of bits requires smaller storage capacity.

I will generate a Pareto curve of my designs. Then, I will compare the FPGA running time with the software running time on FPGA to evaluate the performance of the accelerator.

5. REFERENCES

[1] Sam S Stone, Justin P Haldar, Stephanie C Tsao, BP Sutton, Z-P Liang, et al. Accelerating advanced MRI reconstructions on GPUs. *Journal of parallel and distributed computing*, 68(10):1307–1318, 2008.

[2] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[3] The impact research group. Parboil benchmarks. Available at http://impact.crhc.illinois.edu/parboil/parboil.aspx.

[4] sld-esp Columbia. How to design a signal core SoC. Available at https://www.esp.cs.columbia.edu/docs/singlecore/.