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 plan to explore the design space to get a thorough understanding of the methodology of designing accelerators though ESP.

1. INTRODUCTION

Magnetic Resonance Imaging is commonly used by 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. MRI has scan phase (data acquisition) and image reconstruction phase. Short scan time can increase scanner throughput and reduce patient discomfort, which tends to mitigate motion-related artifacts. High resolution of the image is preferable. Short scan time and high resolution conflict with each other if sampling with the Cartesian trajectory in k-space on a uniform grid, which allows image reconstruction to be performed quickly and efficiently. However, the reconstruction of non-Cartesian trajectory sampling data is faster and less sensitive to imaging artifacts caused by non-Cartesian trajectories, but it increases computation significantly [1]. The computation for MRI-Q matrix is an important algorithm used for image reconstruction with non-Cartesian trajectory sampling [2].

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 MRI-Q matrix through ESP is a good way to learn ESP design methodology.

2. SPECIFICATION

The algorithm for computing MRI-Q matrix is shown in Fig. 1. We want to accelerate the whole computation, mainly by loop unrolling and pipelining the inner for-loop. The programmer view algorithm C code and input dataset come from the Parboil benchmark suit [3]. In this project, we will design accelerators which can accomodate the three datasets provided by Parboil benchmark, and make it capable of dealing with input images with arbitray size.

2.1 Assessment

We aim to design accelerators which can implement Q-matrix computation for any arbitrary input image sizes. The first goal is correctness. The Q-matrix computed by our accelerator should match the Q-matrix computed by software program. We calculate the difference between every output and its golden output, and deem it as a match when the difference is less than a certain threshold, otherwise it is an error. And we also set an error_rate. If it is within a specified small value, we deem that our accelerator meet the correctness goal. The second goal is performance. We will measure the acceleration of our accelerator compared to its software execution running on FPGA board. We want our accelerators

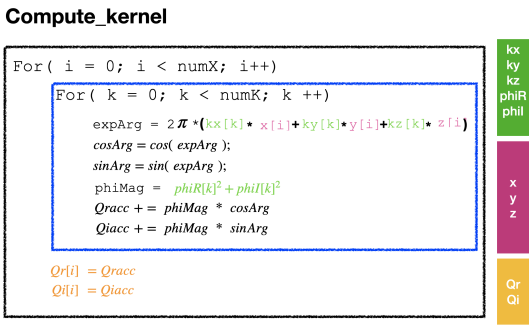

Figure 1: Algorithm for computing MRI Q matrix [1]

can be integrated with both Ariane core and Leon3 core. The generated RTL through Stratus HLS can be integrated and prototyped on FPGA. We will design both baremetal application and Linux application, the accelerator should pass both tests. At last we want to collect speedup data which indicates the acceleration effect. In achieving the second goal, we will do some amount of design space exploration, which includes designing accelerators with different area and latency trade-offs.

2.2 Milestones

1. Analysis of the algorithm and the programmer's implementation in C (by Feb. 19)
2. Learning two tutorials: "How to design an accelerator in SystemC (Cadence Stratus HLS)" and "How to design a single-core SoC"[4][5]. (by Feb. 28)
3. High-Level-Synthesis implementation in SystemC (by Mar. 11)
   - Transform programmer view algorithm to HLS-ready SystemC
   - Initial HLS using the Cadence Stratus tool
4. Bare-metal and Linux application design (by Mar. 18)
5. Evaluation on an FPGA platform (by Mar. 25)
6. Mid-term presentation and report (by Mar. 25)
7. Design Space Exploration (by Apr. 15)
8. Final refinement and analysis (by May 5)
9. Final presentation (May 14) and report (May 15)

2.3 Critical Aspects

1. Use fixed-point datatype and various data conversions
2. Implement sine and cosine functions in SystemC.
3. Try different methods of optimization to reduce latency or decrease area.

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

Table 1: Datasets of MRIQ from Parboil Benchmark

3. PROGRESS

3.1 Programmer View Implementation

3.1.1 Implementation From the Benchmark

The Parboil benchmark provides datasets to run the programmer view implementation. In their github repository [3], we can find documentation of this benchmark. After cloning the whole Parboil to local, under the Parboil folder, I ran the benchmark by the following command:

./parboil run mri-q cpu small

Here, "smallâĂIJ means the dataset is stored under "small" folder, whose input image size is 32*32*32. I have also ran the test for "large", whose input image size is 64*64*64. The image size of the largest dataset is 128*128*128. The bigger the input data size, the higher precision of the reconstructed image.The running time for small and large are around 4.7s and 18.7s, respectively. But for the "128x128x128" input dataset, the run time is 186685s$\sim$ 2.2 days. This implementation was done on the socp03.cs.columbia.edu google cloud server with two cores.

3.1.2 Implementation From Modified C Code

In order to have a comparison between hardware implementation and software implementation, we need to run the software version on the processor of our SoC on FPGA. So I modified the software C code and put this implementation as a function call in the sw/linux/app/mriq.c code.

3.2 Design MRIQ Accelerator in SystemC

The MRIQ application is to compute the Q matrix. Each element of the Q matrix is a complex number with imaginary and real part. The input data can be divided into two groups. One group is coordinates information from frequency space, which is called k-space data. The other group is the sampling positions in 3-D space, which is sometimes called sampling space data. There are 5 variables from k-space, and 3 variables from the sampling space. Each pair of output is computed by using one sampling space point accumulating through the whole k-space. For the "small" and "medium" dataset in Table 1, it has 15 K words at most for k-space data. The storage requirement is 60 K Bytes if data width is 32-bit. The whole k-space data can be stored into PLM. While for the "large" dataset, it is 20 M words, which is 80 M Bytes. For some potential unknown applications with even higher image resolution, the storage requirement for k-space data could be larger. The storage cost may be so high that we want to sacrifice speed. For a specific application, the designer should balance the storage cost and speed requirements. For the MRI-Q matrix application, the Q matrix computation is not in real-time, so we may want to save the cost of storage and compromise on speed. Thus we need at least two architectures to deal with different input image sizes. One is called A0 dealing with small k-space data, the other is A1 dealing with large k-space data.

3.2.1 Configuration Parameters

There are two parameters provided by the benchmark, numX and numK. Since we might deal with one batch data one time, we set four parameters in total: batch_size_x, batch_size_k, num_batch_x, and num_batch_k. And they have the following relationships with numX and numK:

$$numX = batch\_size\_x * num\_batch\_x$$

$$numK = batch\_size\_k * num\_batch\_k$$

$batch\_size\_x$ denotes the size of one batch for sampling space data. num_batch_x indicates how many batches needed to load the whole sampling space data. These four configuration parameters are suitable for both A0 and A1 architecture. While for A0 architecture, num_batch_k is always 1, and $batch\_size\_k$ should be equal to $numK$.

3.2.2 PLM Design

The private local memories of accelerator is generated by "memgen" script provided by ESP. We need a script to tell memgen what kind of PLMs we need. There are three aspects we need to consider when designing PLM for MRI-Q accelerator. Firstly, the word size of the PLM. Input variables from sampling space and output have the same size, while PLMs storing k-space data should have a different word size. Additionally, PLMs storing k-space variables customized for A0 architecture should have size of $numK$, while it should be $batch\_size\_k$ for A1. Secondly, DMA width of the processor core. In order to maximize the latency performance, we can customize ports of PLMs to work with different processor cores. For example, Ariane core has 64-bit DMA width, the PLM should provide two writing ports when loading data into PLM in parallel. while one port is enough for Leon3 core with 32-bit DMA width. Thirdly, parallelism level. In compute phase, how many data we can access in parallel is determined by number of the corresponding PLM ports. For example, if we wanted to do 4 computation in parallel, the PLM which stores the input data should have 4 reading ports. Then we should specify in the hw/mriq_directives.hpp file that what memory we want to use under what conditions.

3.2.3 Data Type

Since the FPGA can not process floating-point data directly, we need to do various data conversions in both the accelerator and the testbench. fpdata.hpp file defines functions used to do datatype conversions, including int2fp(), fp2int(), fp2bv(), bv2fp(). The data conversion flow is concluded in Fig.**??**. In the testbench, data in floating-point type read from files is converted to 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 back to fixed-point representation. After computation finished, The fixed-point data needs to be converted back to sc_int 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.

3.2.4 Functions

There are four functions essential to MRI-Q accelerator design. In load process, *load_data()* function is to load data into different PLMs. This function loads "len" number of words, starting with dma address "dma_addr", into a PLM with name "array". In

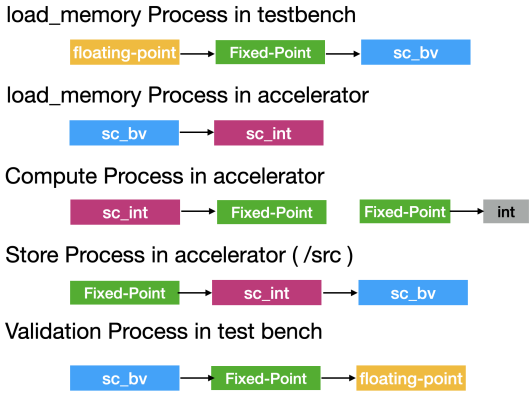

Figure 2: Data conversion in testbench and accelerator

Algorithm 1: Pseudo-code of computeQ() function

```
for (i=0; i < numK; i += unroll_factor) do
    HLS_LOOP_PIPELINE
    for (i = 0; i < unroll_factor; i++) do
        HLS_LOOP_UNROLL
        read from PLMs into registers
          kx[i],ky[i],kz[i],phiR[i],phiI[i]
    end
    for (i = 0; i < unroll_factor; i++) do
        HLS_LOOP_UNROLL
        compute Qracc_p[i], Qiacc_p[i]
    end
    for (i = 0; i < unroll_factor; i++) do
        HLS_LOOP_UNROLL
        Qracc += Qracc_p[i]
        Qiacc += Qiacc_p[i]
    end
end
```

this function, dma_info is configured first, including dma_addr, dma_length, and dma_size. Then it stores data from dma_read_chnl to PLM "array".

$$\text{load\_data}(\text{array}[], \text{dma\_addr}, \text{len})$$

The second function is *store_data*, which is the counterpart of load_data(). It sends data stored in "array" PLM through dma_write_chnl to the testbench.

$$\text{store\_data}(\text{array}[], \text{dma\_addr}, \text{len})$$

The third function is *computeQ* which is to compute a pair of output data with the whole k-space variables and one sampling space data point (x, y, z). In computeQ, there are three tasks being pipelined. The first task is reading data from PLM into registers kx, ky, kz, phiR, and phiI. The second task is to do the computation part. The third taks is to accumulate the intermediate results. After the loop finishes, it stores the accumulated results into two memory addresses pointed by pointers Qr and Qi.

$$\text{computeQ}(\text{x}, \text{y}, \text{z}, \text{batch\_size\_k}, \text{pingpong\_x}, \text{sin\_table}, *\text{Qr}, *\text{Qi})$$

The fourth function is *mySine* which is to compute the sine value of an input.

$$\text{mySinf}(\text{angle}, \text{sin\_table})$$

The algorithm in Fig.1 shows that it needs to compute triangular functions, sine and cosine. Since Stratus HLS doesn't provide sine and cosine function, we implemented a sine function *mySine*. We firstly convert any input value to 0 $\pi/2$ range, x, then find the closest data point to x in the sin_table, then do interpolation to get sin(x).

3.2.5 *Processes*

In the main hardware design source file mriq.cpp, we have three processes, load_input, compute_kernel, and store_output. load_input handshakes with compute_kernel. compute_kernel handshakes with store_output. These three processes are pipelined in the background. For both A0 and A1 architectures, we are utilizing ping-pong buffers to store sampling space variables (x, y, z) and (Qr, Qi). A flag pingpong_x is to tell each processes whether we should read from or write to plm_var_ping or plm_var_pong memory. For A1 architecture, we also need ping-pong buffers to store k-space variables. Thus we need another flag pingpong_k as a local variable in each process. Conditional loading in the load_input process is implemented. We have a counter variable counting how many times load_input process has been implemented. Whenever we load one batch of sampling space variables, we need to load k-space variables for $num\_batch\_k * batch\_size\_x$ times, and we flip pingpong_k flag and handshake with compute_kernel process for every loading. Then we flip pingpong_x and reset the counter in load_input process. After the computation of the current batch of sampling space variables finishes, we flip pingpong_x flag once and handshake with store_output process in compute_kernel process.

3.3 Testbench Design

The ESP provides skeleton code for the testbench design. We only need to fill in customized code in each function. In tb/system.hpp file, it defines a class system_t. In this class declaration, there is a constructor function, and other members. For example, configuration parameters to the accelerator, variables used by different methods, and methods. Here we declare an additional method used by load_memory function, which is load_mem() method. In system.cpp file, we have four methods definitions. In load_memory(), we initialize the necessary parameters first, then fill the "mem" variable with input data and fill the "gold" pointer with golden output data. Since the accelerator can only deal with fixed-point data, we convert floating-poing data to sc_bv to be transported to the accelerator. In dump_memory(), we convert and store the output sent from the accelerator to a pointer variable "out" with floating-point type. In validate(), we compare "out" with "gold" to verify the accelerator design.

We define some helper functions in a separate folder, "common" folder. In the testbench design, we used init_parameter() and validate_buffer(), defined in utils.h file. These two helper functions are also used in software testing application code.

3.4 Software Testing Applications

ESP enables two testing on FGPA. One is bare metal application testing and the other is testing the accelerator with Linux OS. These two applications are in sw/ folder.

For the bare metal application, we need to fill in code in init_parameters(), init_buf() and validate_buf(). Since bare metal implementation doesn't support many C librabries, we can't use init_buffer function in common/utils.h file. We generated testing input data beforehand

and included it as a file. While in validate_buf function, we first convert output sent from the accelerator from fixed-point data to floating-point data by using fixed32_to_float() function provided by fixed_point.h. Then use validate_buffer() function from common/utils.h to validate the output.

For linux application, we also only need to fill customized code in init_parameters(), init_buf(), and validate_buf() function, same as the bare-metal application except the init_buf() function. Since we run the test program on linux OS, the init_buf can use the function init_buffer() in common/utils.h, which reads input data from file. We can also test the execution time of software program running on the processor tile of SoC which is prototyped on the FPGA. When running the linux application, we pass three arguments: name of input File, name of output file, and the answer to whether we want to run software program with "0" indicating "no" and "1" indicating "yes".

3.5 Testing Results

The testbench can help us debugging our accelerator at design phase. When we don't have "#if" directives, we can run behavioral simulation by running make target "make mriq_stratus-exe". This will check if our accelerator has the expected behavior. Then we can generate RTL with "make mriq_stratus-hls" and simulate RTL with the testbench. When we use "#if" directive to generate multiple RTL implementations, the above three make targets won't work correclty. Then we need to use the debugging method provided on the ESP tutorial, which is to simulate one specific RTL with "make debug_<RTL name>". For example, if we want to test A1 architecture with DMA width as 64 and parallelism level as 4, we can run "make debug_BASIC_P4_A1_DMA64_V". The running result is shown in Fig.3.

```
ncsim> Info: dma-controller: reset done

Info: testbench: reset done

Info: testbench: load memory completed

Info: testbench: config done

Info: testbench: @310 ns BEGIN - mriq

Info: testbench: @4330 ns END - mriq

Info: testbench: dump memory completed

Info: testbench: validation passed!
```

Figure 3: RTL simulation (BASIC_P4_A1_DMA64_V)

After we finish the software testing program design, we can also simulate and test the bare metal application, and run the Linux application after booting the FPGA with Linux image. For the baremetal application, for example, we test "P4_A0_DMA64" RTL on FPGA, the result is shown in Fig.4. If we generated Linux image and registered the mriq_stratus accelerator as a device driver, we can test the linux app on FGPA. The running result is shown in Fig.5. Fig.5-(a) shows that we also ran the software program on the SoC core, and the execution time is roughly 10 times slower than the hardware accelerator on a small testing data.

4. DESIGN SPACE EXPLORATION

After we fixed the basic architectures of the design. We can try different HLS knobs to optimize our accelerator from area and latency directions.

```
ESP-Ariane first-stage boot loader

Scanning device tree...
[probe] sld,mriq_stratus.0 registered
        Address   : 0x60010000
        Interrupt : 6
**************** sld,mriq_stratus.0 ****************
  memory buffer base-address = 0xa0100b60
  ptable = 0xa0100d10
  nchunk = 1
  --------------------
  Generate input...
  -> Non-coherent DMA
  Start...
  Done
  validating...
  ... PASS
```

Figure 4: Running bare metal app on FPGA

```
# ./mriq_stratus.exe test_small.bin test_small.out 1

====== mriq_stratus.0 ======

  .num_batch_k = 1
  .batch_size_k = 16
  .num_batch_x = 1
  .batch_size_x = 4

  ** START HW TESTING **
  > Test time: 17960560 ns
    - mriq_stratus.0 time: 915960 ns

  ** DONE **
+ HW Test PASSED

  ** START SW TESTING **

  > Test time : 9316280 (ns)

  ** Done! **

+ SW Test PASSED!

====== mriq_stratus.0 ======
```

(a) w/ running sw

```
# ./mriq_stratus.exe test_small.bin test_small.out 0

====== mriq_stratus.0 ======

  .num_batch_k = 1
  .batch_size_k = 16
  .num_batch_x = 1
  .batch_size_x = 4

  ** START HW TESTING **
  > Test time: 16157840 ns
    - mriq_stratus.0 time: 2462400 ns

  ** DONE **
+ HW Test PASSED

====== mriq_stratus.0 ======
```

(b) w/o running sw

Figure 5: Running Linux app on FPGA

4.1 Low Area Optimization

4.1.1 Customize PLM

We want to customize PLM size for two different input variables. k-space variables have the same size while sampling-space variables have the same size but different from the k-space variables. So we set different PLMs for these two groups.

Since we can customize number of ports for each PLM, we shouldn't waste any ports in order to save area. For PLMs storing input data, we set the number of writing ports accordingly to be consistent with DMA width. For example, if DMA width is 64, while data width is 32, then there are two writing ports. If DMA width and data width are both 32, then we only need one writing ports. The number of reading ports of different PLMs depends on the parallelism level. If parallelism level is 4, we need to read 4 k-variables in parallel in the compute_kernel process. So 4 reading ports are needed.

4.1.2 Fixed-Point Datatype Optimization

The MRI-Q accelerator deals with fixed point datatype. We can reduce fixed point precision to reduce both latency and area. When we have only one type of fixed-point data representation and reduce the word length to 29, there are no errors in output data and both area and latency reduced. Further reduction in word length led to output errors. In order to further reduce the data width of the fixed-point representation, I analyzed all the testing data provided by the Parboil benchmark suite. Some data are bound to be very small value, for example, the output of trigonometric functions is always less than 1, and some are large values. The range of data are shown

in Table.3. I set two different fixed-point representations, shown in Table.2. We use FPDATA_S to represent kx, ky, kz, phiR, phiI, x, y, z, and phiMag, sinArg, cosArg. And we use FPDATA_L to represent the other variables. The result shows that the error rate is 0.17% when error threshold (percentage difference between computed output and golden output), is set as 5%. So the data width can be as small as 24 compared to 32.

	WL	IL	Fractional Bits
FPDATA_S	24	5	19
FPDATA_L	24	11	13

Table 2: Fixed-point data representation

Variables	Max	Min	Integer bits
x	3.083235	-0.500000	3
y	0.484375	-0.500000	1
z	0.484375	-0.500000	1
kx	10.889303	-10.889303	5
ky	16.019472	-16.019472	5
kz	0.585179	-0.585179	1
phiI	0.484375	0.312500	0
phiR	0.484375	0.406250	0
phiMag	0.469238	0.262695	0
cosArg	1.000000	-1.000000	2
sinArg	1.000000	-1.000000	2
expArg	226.168869	-226.168869	9
Qracc	961.000000	-472.693146	11
Qiacc	97.710320	-97.710320	8

Table 3: Value range for all variables

We experimented with small data set and we extracted area from the synthesized RTL and extracted latency data from RTL simulation. This small dataset has numX as 4 and numK as 16, and it is running with accelerator A0_P4_DMA64. The extracted area and latency data are shown in Table.4. Area reduces by 31.5%. Latency also reduces by 11.49%. But when we simulate the bare-metal application, it doesn't support data width as 24. So we set FPDATA_S and FPDATA_L with the same specification (WL, IL) = (32, 12) in the other evaluations.

	$WL = 32$	$WL = 24$	Gain
area	0.1185	0.0812	-31.50%
latency (ns)	1740	1540	-11.49%

Table 4: Performance of FP precision reduction

4.2 Low Latency Optimization

We have two directions to reduce latency. The first one is to accelerate the loading and storing process. The second one is to accelerate the computation. Since the data transportation between processor and the accelerator is done in serial, we can increase the burst rate and DMA width. In our case, burst rate is fixed and the DMA width is determined by the processor core we have in our SoC. The Ariane core provides DMA width as 64-bit, while Leon3 core has DMA width as 32-bit. In the computation phase, we can do loop unrolling and loop pipeline to accelerate. Since we don't have the choice to work on accelerating load and store process, we can only work on the compute_kernel.

A0 architecture loads all the k-space data into PLM. It uses ping pong buffer to load sampling space data. The RTL simulation waveform of A0_P4_DMA64 shows that loading one batch of sampling space data takes 2 μs with batch_size_x set as 128, and generating one pair of output data takes 7 μs . Thus generating 128 pairs of output data takes 900 μs. So the compute_kernel process is dominating other processes, and the loading process of the sampling space data is totally hidden by computation process. It also means that the acceleration done on compute_kernel will have the biggest impact on the final latency. But A1 architecture behaves very differently. When we set batch_size_k as 1024, loading five k-space variables takes 25 μs. If num_batch_k was 10, then preparing all the input to generate one pair of output takes 25μs $*10 = 250\mu$s, while the computation takes 7us to finish. Loading process is dominating for A1 architecture and any optimization to the compute_kernel won't work.

First, let's look at the Pareto curve of A0 architecture, which is shown in Fig.6. The area and latency data is extracted from RTL synthesis results and RTL simulation results. The testing parameters are shown in Table 5. The number in P-<number> indicates the unroll-factor. We can see that latency is decreasing when we increase the parallelism level. We have tried unroll factor 4, 8, 16. In the compute_kernel process, we have three tasks which can be pipelined. The first one is to read data from PLMs to registers. The second task is to do computation, and the third task is to store the computed results. For our MRI-Q application, the third step is to accumulate the intermediate results. Every task can be unrolled for unroll-factor number of times. In order to support reading from the same PLM for unroll-factor number of data into registers in parallel , the PLM in question should have unroll-factor number of reading ports.

Configurable variables	A0	A1
numX	256	256
batch_size_x	128	128
num_batch_x	2	2
numK	3072	3072
batch_size_k	3072	1024
num_batch_x	1	3

Table 5: Testing Parameters of the Pareto curve

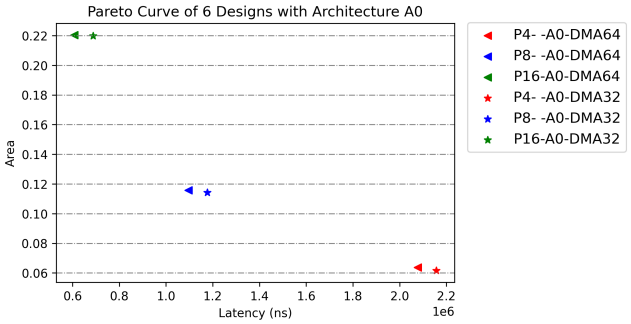

Figure 6: Pareto curve of 6 designs of A0

The latency optimization for A1 architecture should solely depend on the DMA width as we have discussed above. The optimization done in compute_kernel shouldn't have any effect, shown in Fig.7.

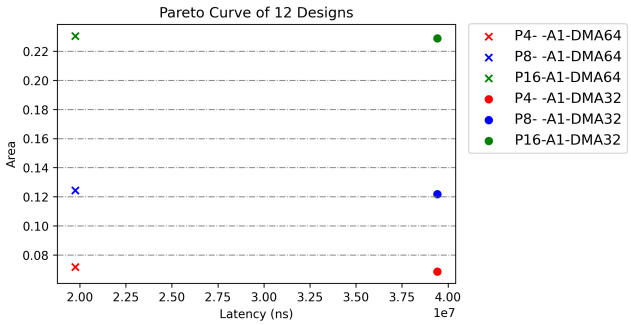

Figure 7: Pareto curve of 6 designs of A1

4.3 Results of Testing on FPGA

By invoking linux on FPGA, we collected both the software execution time running on the Ariane core and the execution time running on the MRI-Q accelerator. For A0 architecture, data is shown in Fig.8 for 32x32x32 (D32) and 64x64x64 (D64) dataset. The right y-axis shows the speed-up of the two testing. The speed up is greater than 5000 for both datasets when parallelism level is 16. The testing result of D32 shows a slightly higher speed-up than D64 is because that numK of D32 is 1.5X of D64. Increasing parallelism in compute can improve latency, same as the extracted data from the synthesized RTL.

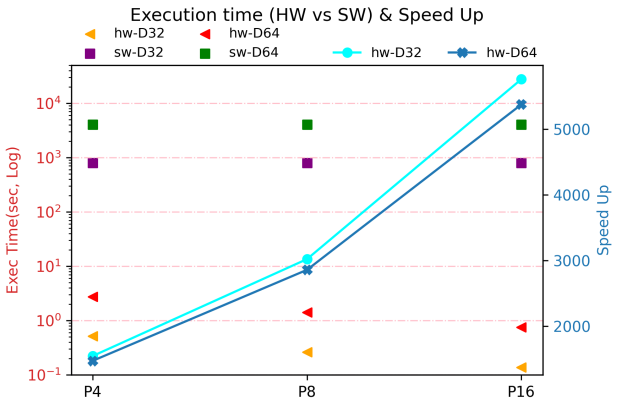

Figure 8: Latency comparison of D32 and D64 Dataset

For A1 architecture, latency results are shown in Fig.9 and Fig.10. For 128x128x128 dataset, numX = numK = 2048K. To evaluation the speedup of accelerator with A1 architecture, I tested P4_A1_DMA64. As we discussed in section 3.3, parallel level in compute process won't affect latency performance, so we choose the P4 accelerator. (It is possible that there is a parallelism level which is smaller than 4 and can also balance the loading and computing processes to further reduce area while with the same latency performance.).The speed-up value is increasing when increasing the data size. Fig.9 shows the execution time comparison of software and hardware and speedup when numK is fixed at 4K and numX is increasing. Fig.10 shows how execution time and speedup change over numK increasing from 4K to 256K while numX is fixed as 4. At some point, speedup saturates when numK and numX increases. For our A1 architecture, we get at least 400 times acceleration compared with software execution for any arbitrary input image size.

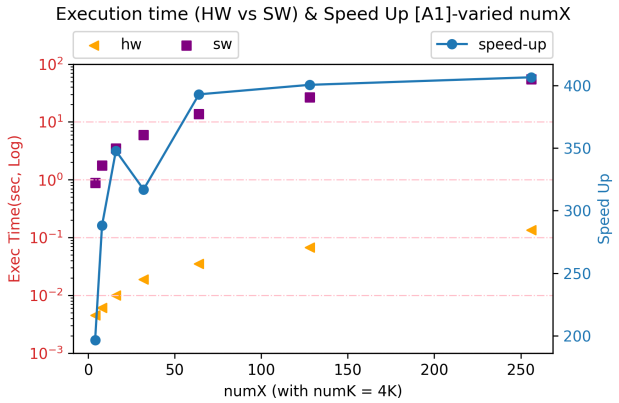

Figure 9: Latency with varying numX (numK=4K)

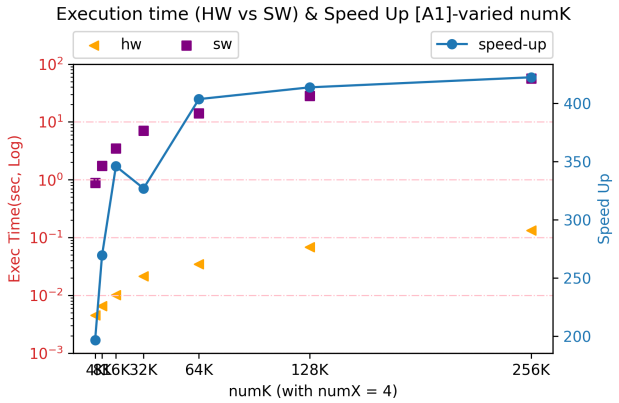

Figure 10: Latency with varying numK (numX=4)

5. CONCLUSION

In this project, I have designed a set of accelerators using ESP to compute the Q matrix for MRI image reconstruction. Any arbitrary input data sizes can find an accelerator to work with. The accelerators are tested on the FPGA through both bare-metal application and Linux application. I have also done different levels of design space exploration. Customizing PLM designs and reducing the fixed point precision can reduce both area and latency. Loop-unrolling and loop-pipelining in the main computation-intensive loop can reduce latency proportionally when compute kernel is dominating the whole procedure.

6. LIMITATIONS OF THE WORK

We don't compare our acceleration result with the state-of-the-art MRI image recustruction performance since we only accelerate one part of the iamge reconstruction. The whole image recustruction of non-Cartician scanning contains three steps: computing Q-matrix, computing the vector $\mathbf{F}^H\mathbf{d}$, and finding the image iteratively via a conjugate gradient linear solver. The first two steps are very similiar. [1] Since our accelerator only accelerate the first step, we can't evaluate the reconstructed image quality and the performance with other works. Also, we don't know the error tolerance of the first step.

7. POTENTIAL FUTURE WORK

- Design the same set of accelerators with Vivado HLS C/C++

through ESP for the MRI-Q matrix computation. Compare the latency and area performance between the generated RTLs through two different design flows.

- Design accelerators to compute the second step mentioned in Section 6.

8. 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/`.

[5] sld-esp Columbia. How to design an accelerator in SystemC. Available at `https://www.esp.cs.columbia.edu/docs/systemc_acc/`.